Memory Virtualization

3) Dynamic Relocation

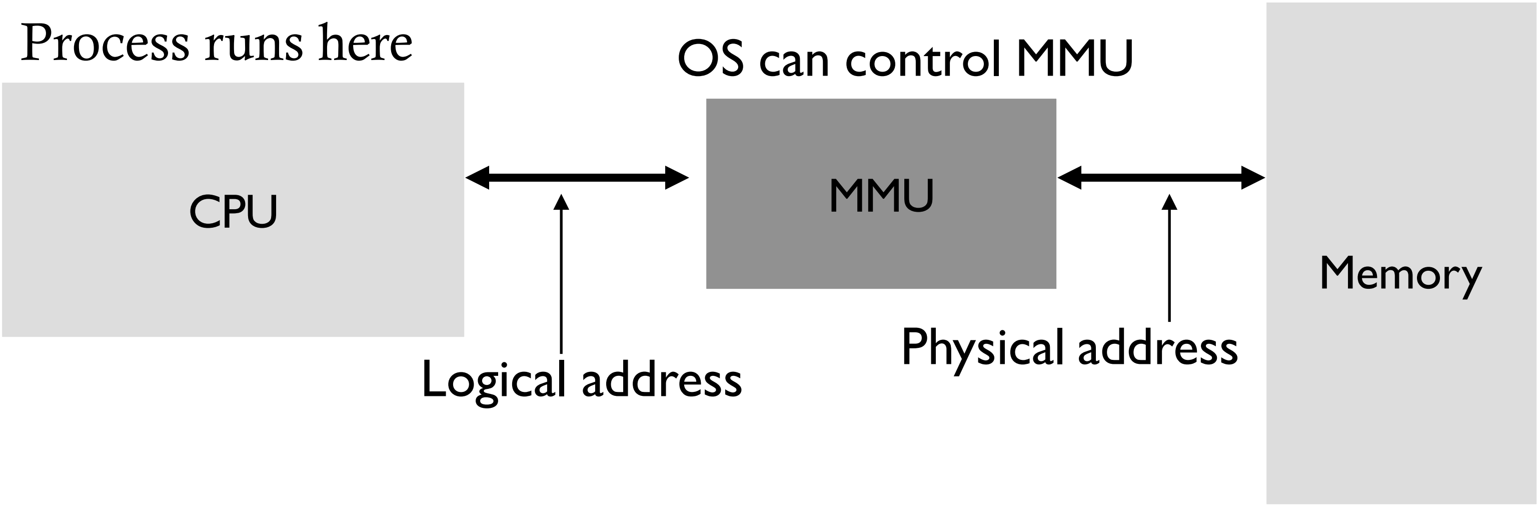
Goal: Protect processes from one another

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



Hardware support for Dynamic Relocation

Two operating modes

- Privileged (protected, kernel) mode: OS runs
  - When enter OS (trap, system calls, interrupts, exceptions)
  - Allows certain instructions to be executed
    - **Can manipulate contents of MMU**
  - **Allows OS to access all of physical memory**
- User mode: User processes run
  - **Perform translation of logical address to physical address**

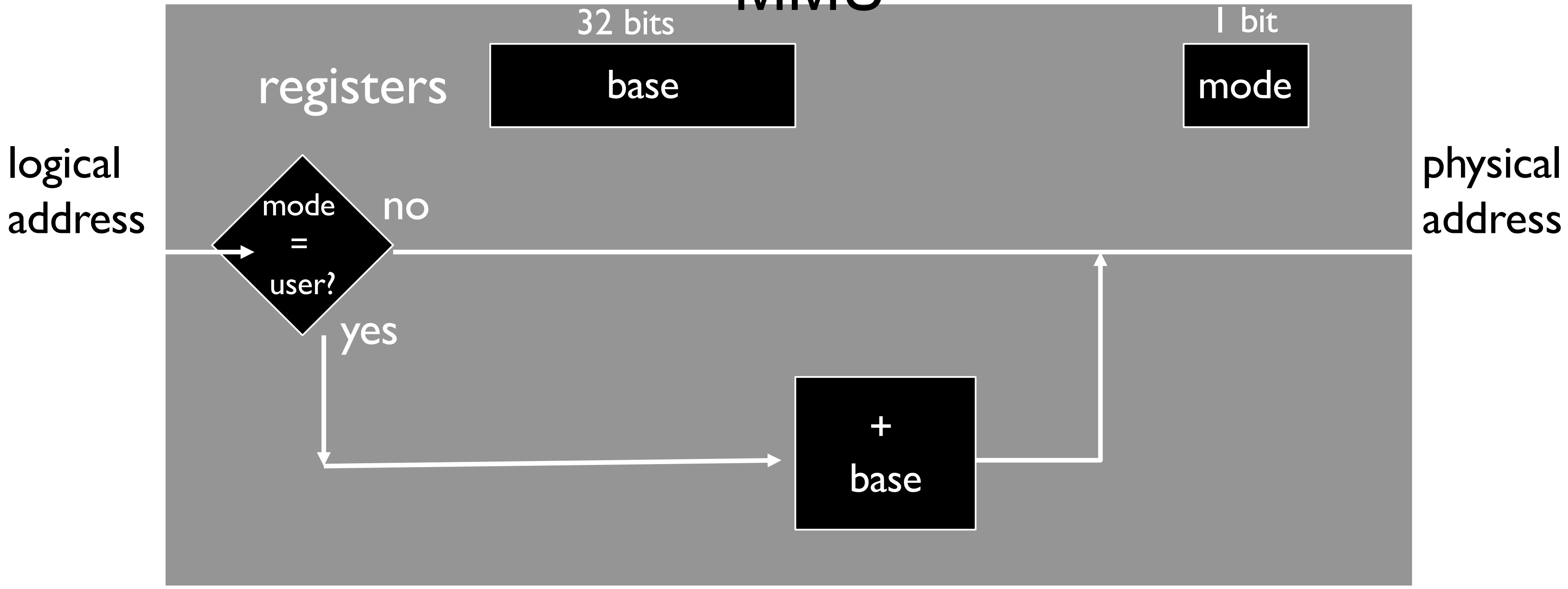
A minimal MMU contains **base register** for translation

- base: start location for address space

Implementation of Dynamic Relocation: BASE REG

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

- MMU

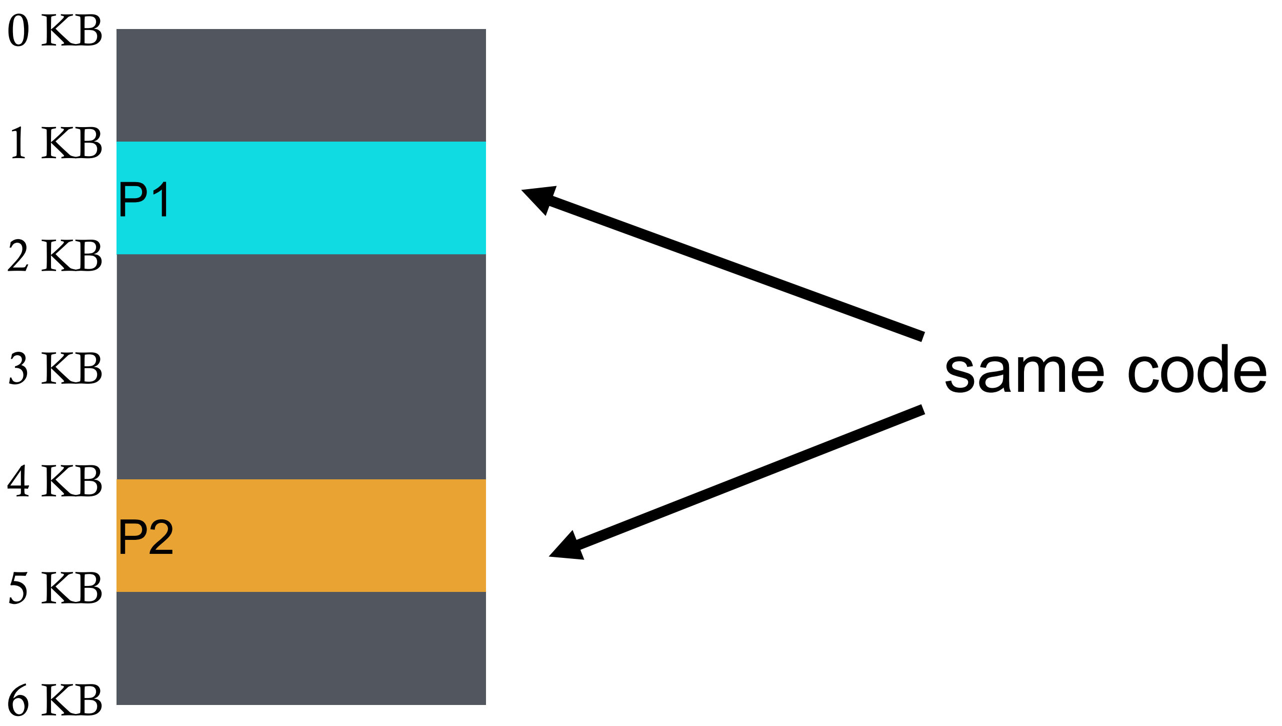


Dynamic Relocation with Base Register

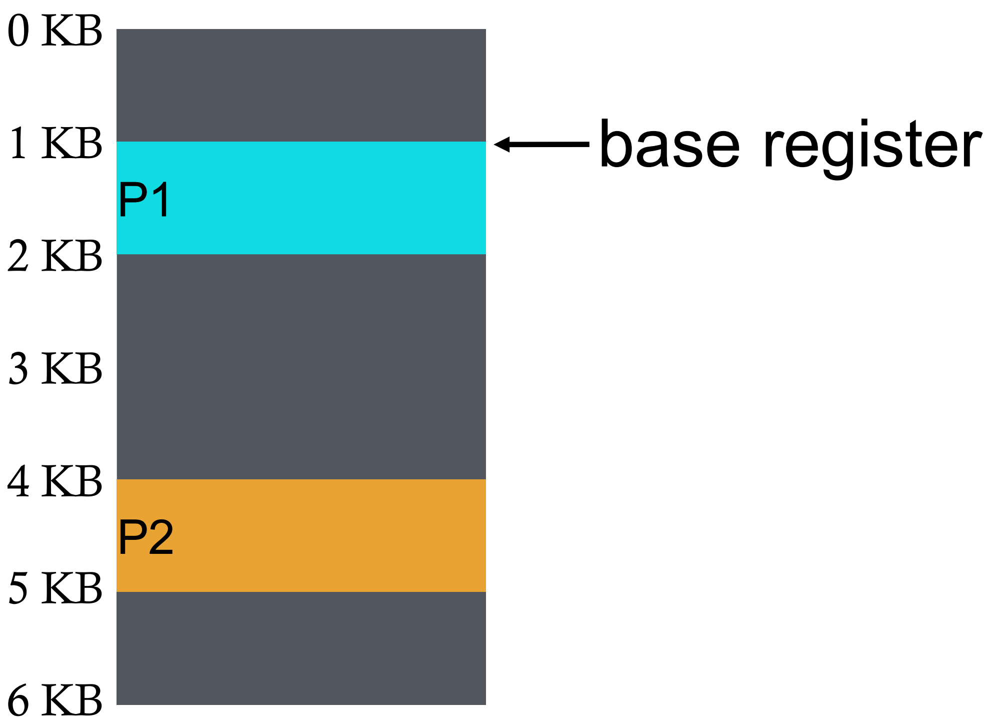
Idea: translate virtual addresses to physical by adding a fixed offset each time.

Store offset in base register

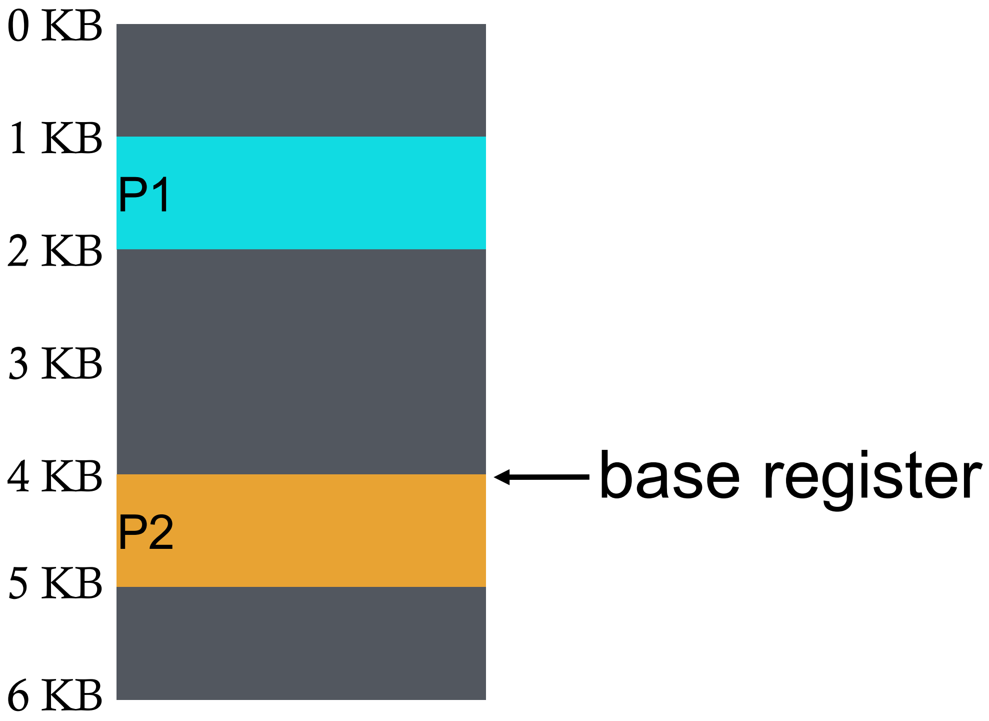
Each process has different value in base register



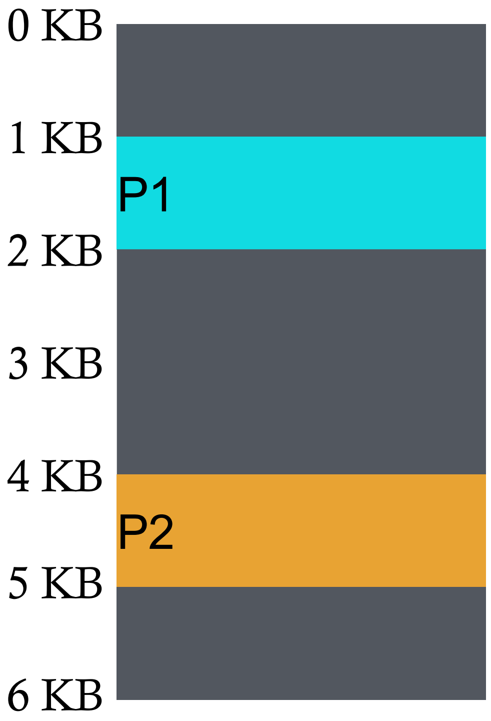
VISUAL Example of DYNAMIC RELOCATION:
BASE REGISTER



P1 is running

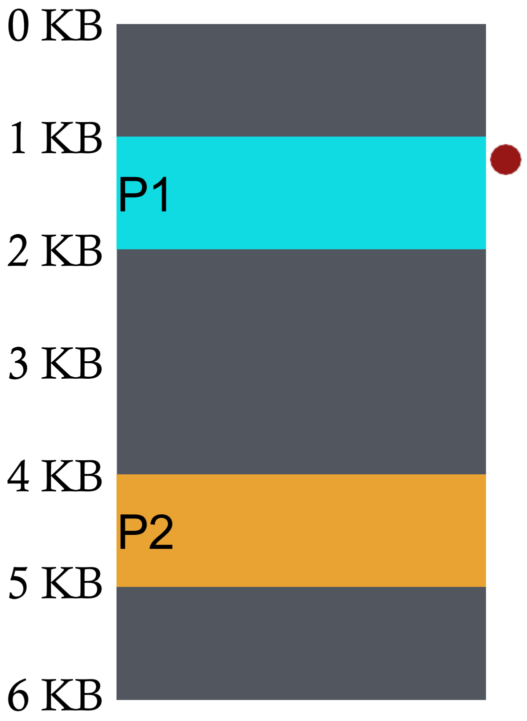


P2 is running

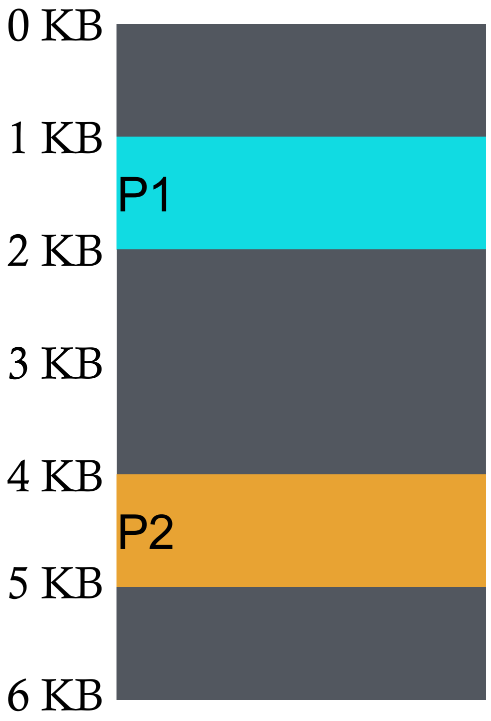


(Decimal notation)

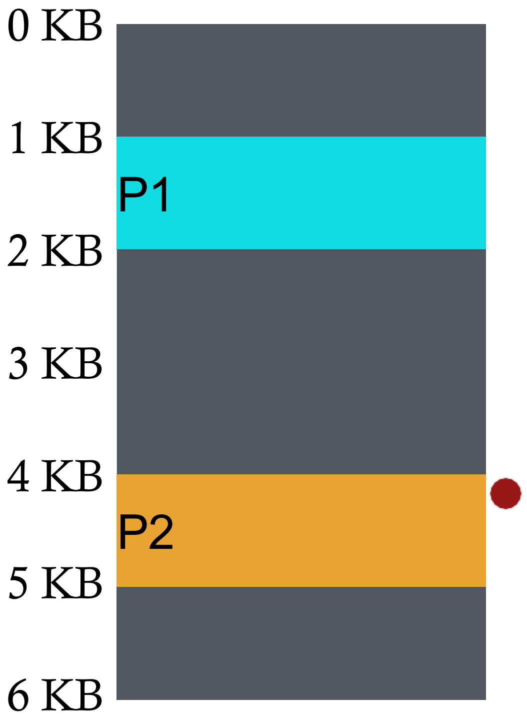
Virtual	Physical
P1: load 100, R1	



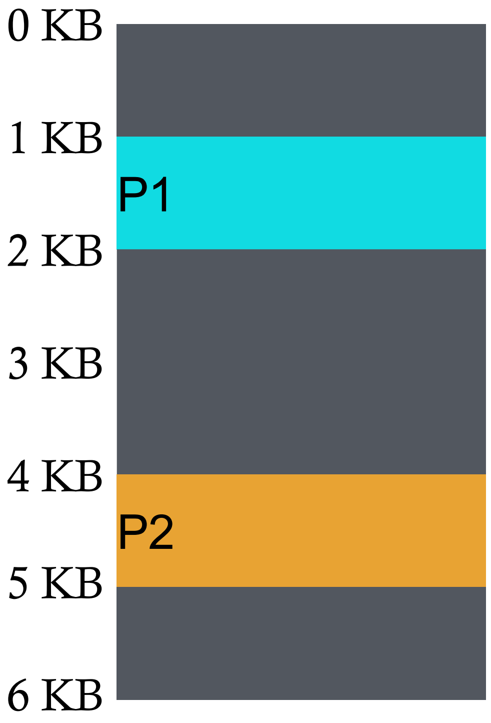
Virtual	Physical	
P1: load 100, R1	load 1124, R1	(1024 + 100)



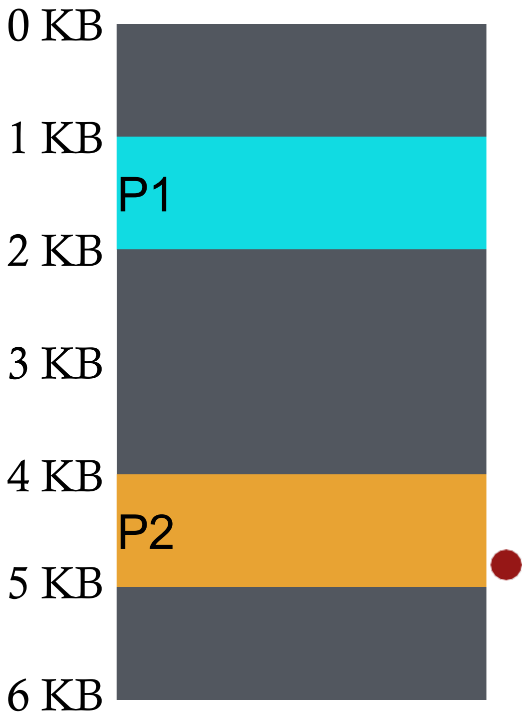
Virtual	Physical
P1: load 100, R1	load 1124, R1
P2: load 100, R1	



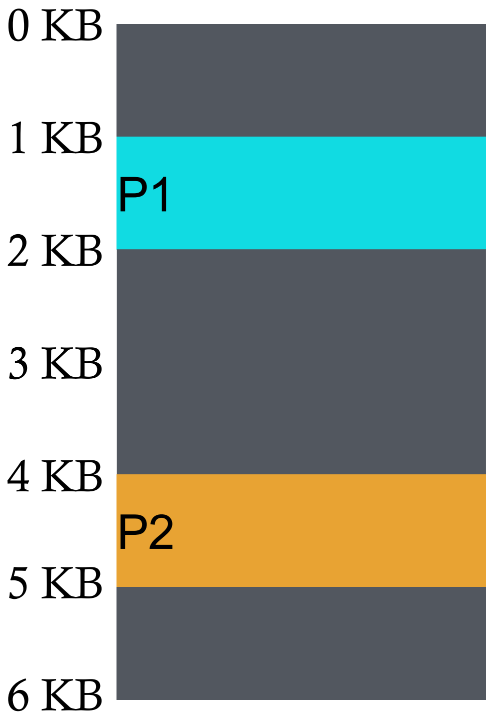
Virtual	Physical	
P1: load 100, R1	load 1124, R1	
P2: load 100, R1	load 4196, R1	(4096 + 100)



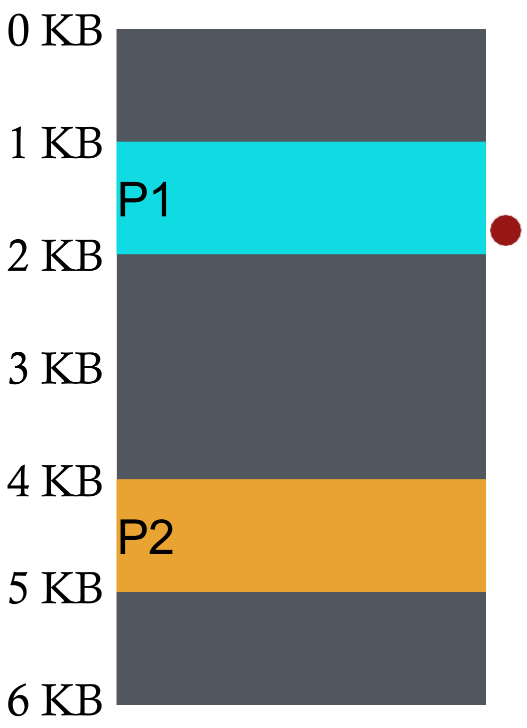
Virtual	Physical
P1: load 100, R1	load 1124, R1
P2: load 100, R1	load 4196, R1
P2: load 1000, R1	



Virtual	Physical
P1: load 100, R1	load 1124, R1
P2: load 100, R1	load 4196, R1
P2: load 1000, R1	load 5196, R1



Virtual	Physical
P1: load 100, R1	load 1124, R1
P2: load 100, R1	load 4196, R1
P2: load 1000, R1	load 5196, R1
P1: load 1000, R1	

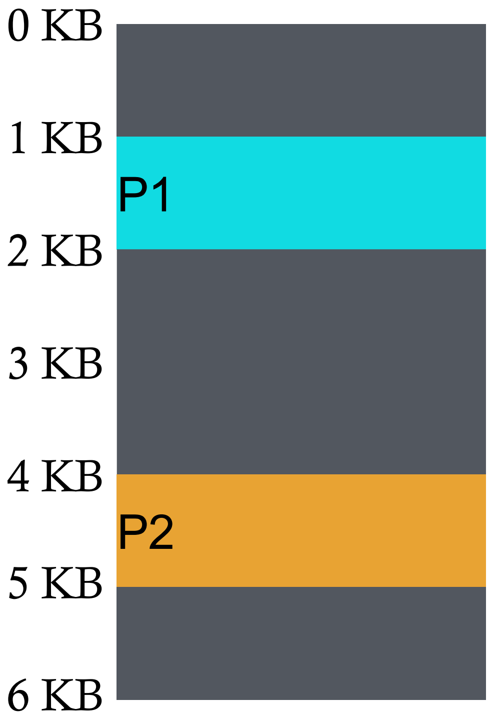


Virtual	Physical
P1: load 100, R1	load 1124, R1
P2: load 100, R1	load 4196, R1
P2: load 1000, R1	load 5196, R1
P1: load 1000, R1	load 2024, R1

Who Controls the Base Register?

What entity should do translation of addresses with base register?
(1) process, (2) OS, or (3) HW?

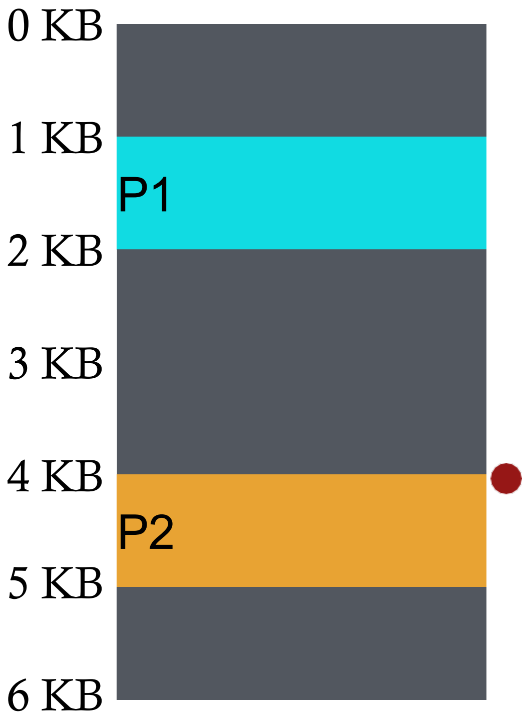
What entity should modify the base register?
(1) process, (2) OS, or (3) HW?



Virtual	Physical
P1: load 100, R1	load 1124, R1
P2: load 100, R1	load 4196, R1
P2: load 1000, R1	load 5196, R1
P1: load 1000, R1	load 2024, R1

Can P2 hurt P1?
Can P1 hurt P2?

Does the base register mechanism protect processes from each other?



Virtual	Physical	
P1: load 100, R1	load 1124, R1	
P2: load 100, R1	load 4196, R1	
P2: load 1000, R1	load 5196, R1	
P1: load 1000, R1	load 2024, R1	
P1: store 3072, R1	store 4096, R1	(3072 + 1024)

Can P2 hurt P1?

Can P1 hurt P2?

Does the base register mechanism protect processes from each other?

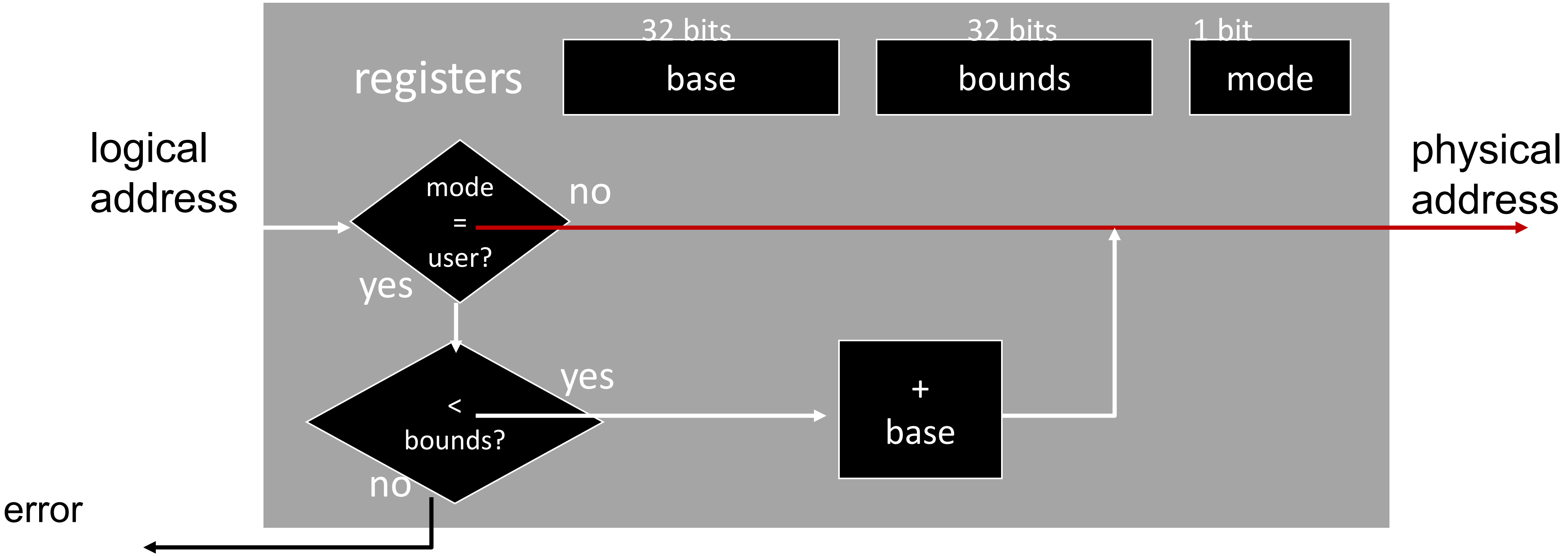
4) Dynamic with Base+Bounds

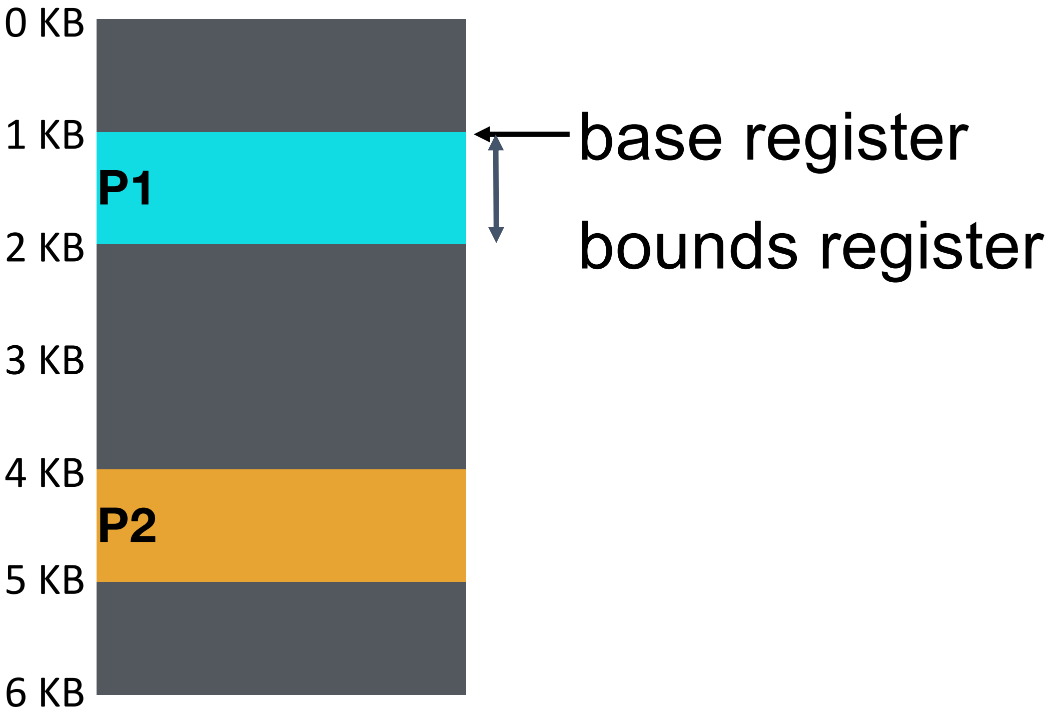
- Idea: limit the address space with a bounds register
- Base register: smallest physical addr (or starting location)
- Bounds register: size of this process’s virtual address space
  - Sometimes defined as largest physical address (base + size)
- OS kills process if process loads/stores beyond bounds

Implementation of BASE+BOUNDS

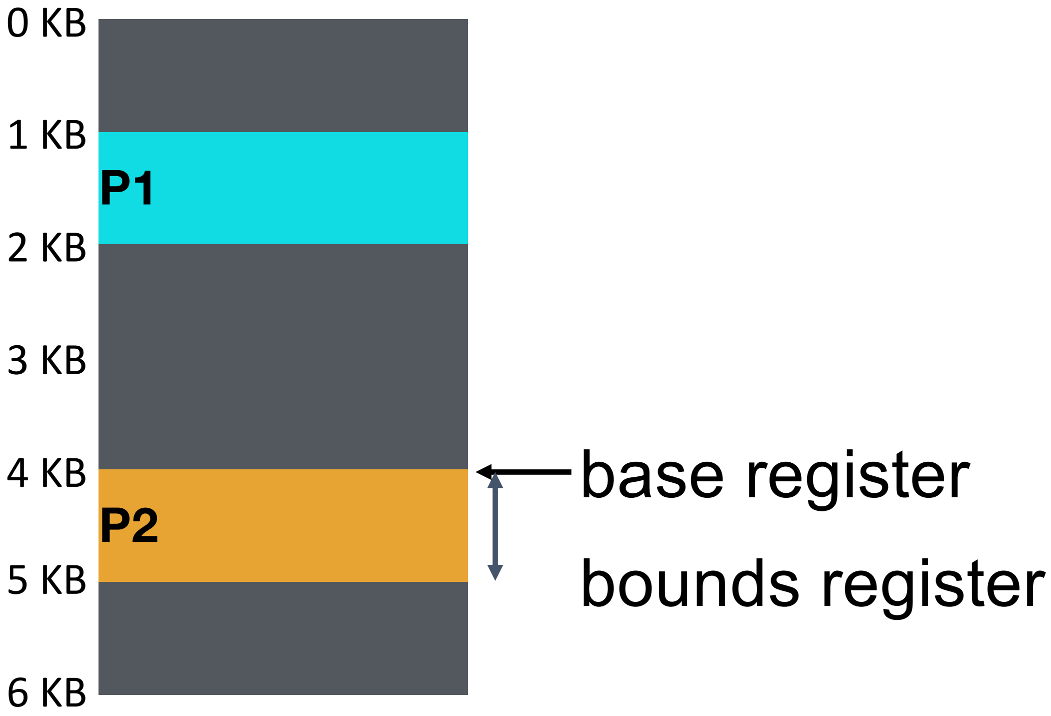
Translation on every memory access of user process

- MMU compares logical address to bounds register
  - if logical address is greater, then generate error
- MMU adds base register to logical address to form physical address





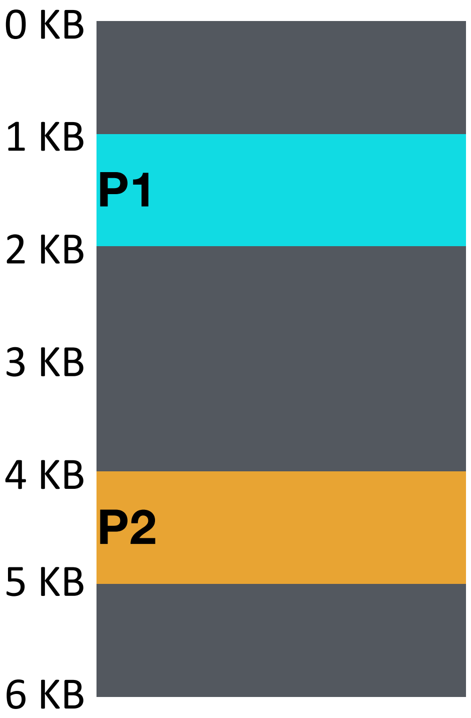
P1 is running



base register

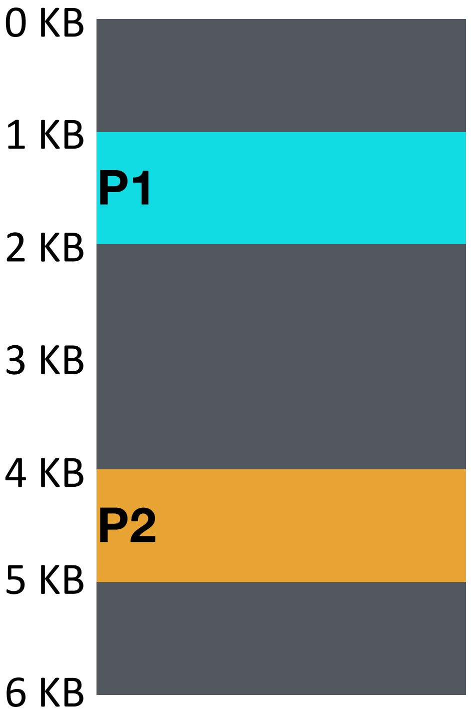
bounds register

P2 is running



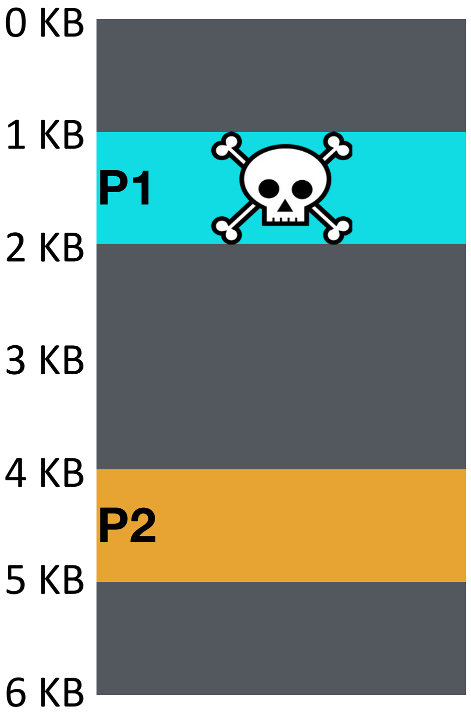
Virtual	Physical
P1: load 100, R1	load 1124, R1
P2: load 100, R1	load 4196, R1
P2: load 1000, R1	load 5196, R1
P1: load 1000, R1	load 2024, R1
P1: store 3072, R1	

Can P1 hurt P2?



Virtual	Physical	
P1: load 100, R1	load 1124, R1	
P2: load 100, R1	load 4196, R1	
P2: load 1000, R1	load 5196, R1	
P1: load 1000, R1	load 2024, R1	
P1: store 3072, R1	**interrupt OS!**	3072 > 1024

Can P1 hurt P2?



Virtual	Physical
P1: load 100, R1	load 1124, R1
P2: load 100, R1	load 4196, R1
P2: load 1000, R1	load 5196, R1
P1: load 1000, R1	load 2024, R1
P1: store 3072, R1	**interrupt OS!**

Can P1 hurt P2?

Managing Processes: Base & Bounds

Context-switch

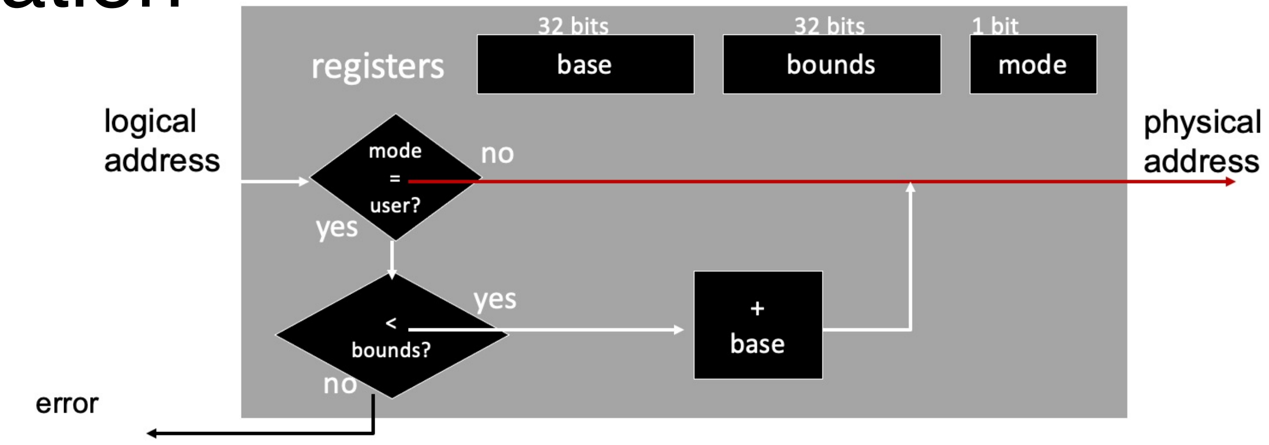
- Add base and bounds registers to Process Control Block
- Steps
  - Change to privileged mode
  - Save base and bounds registers of old process
  - Load base and bounds registers of new process
  - Change to user mode and jump to new process

Protection requirements

- User process cannot change base and bounds registers
- User process cannot change to privileged mode

Base and Bounds Advantages

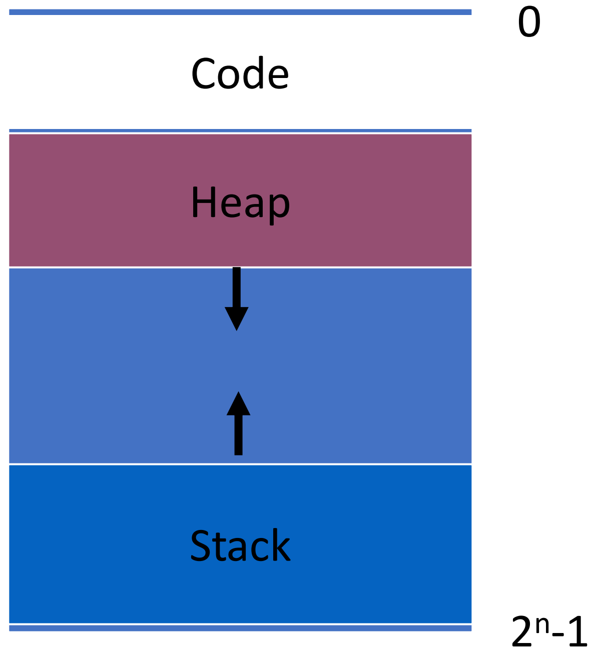
- Provides protection (both read and write) across address spaces
- Supports dynamic relocation
  - Can place process initially at locations different from assumed in the program code
  - Also, move address spaces later if needed
- Simple, inexpensive implementation
  - Few registers, little logic in MMU
- Fast
  - Add and compare in parallel



Base and Bounds DISADVANTAGES

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

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



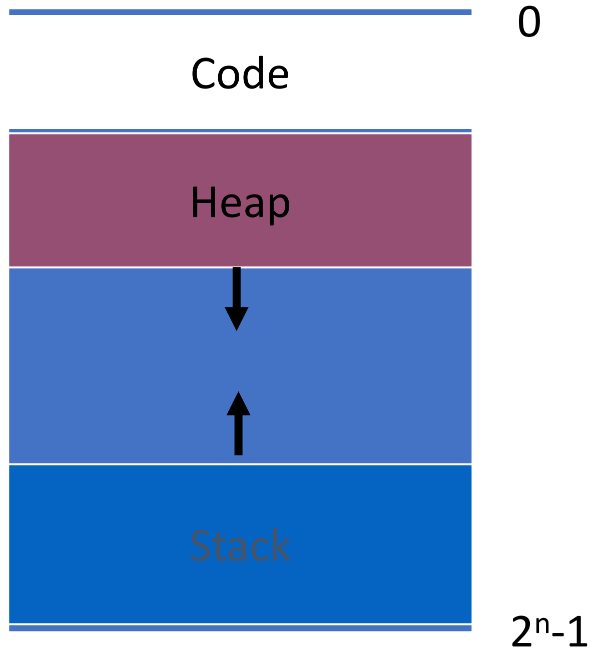
5) Segmentation

Divide address space into logical segments

- Each segment corresponds to logical entity in address space
  - code, stack, heap

Each segment can independently:

- be placed separately in physical memory
- grow and shrink
- be protected (separate read/write/execute bits)



Segmented Addressing

Process now specifies segment and offset within segment

How does process designate a particular segment?

- Use part of logical (virtual) address
  - High-order bits of logical address select segment
  - Low-order 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

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

Translate logical addresses (in hex) to physical addresses

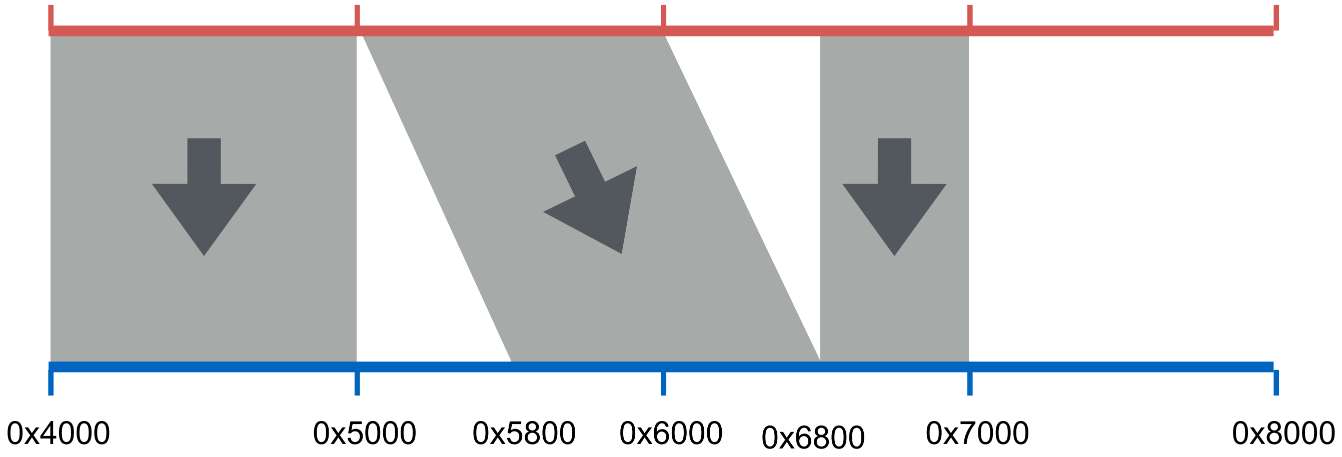
0x0240:

0x1108:

0x265c:

0x3002:

Assume 14-bit virtual addresses, high 2 bits indicate segment

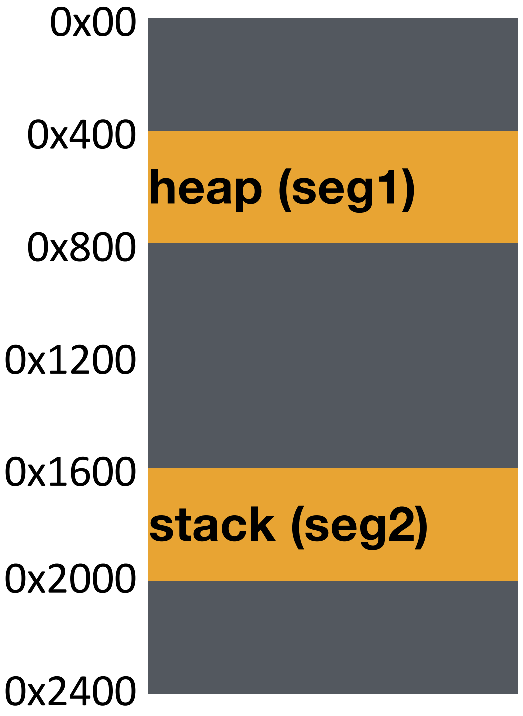


Where does segment table live?

All registers, MMU

0x4000	0xfff
0x5800	0xfff
0x6800	0x7ff

Visual Interpretation



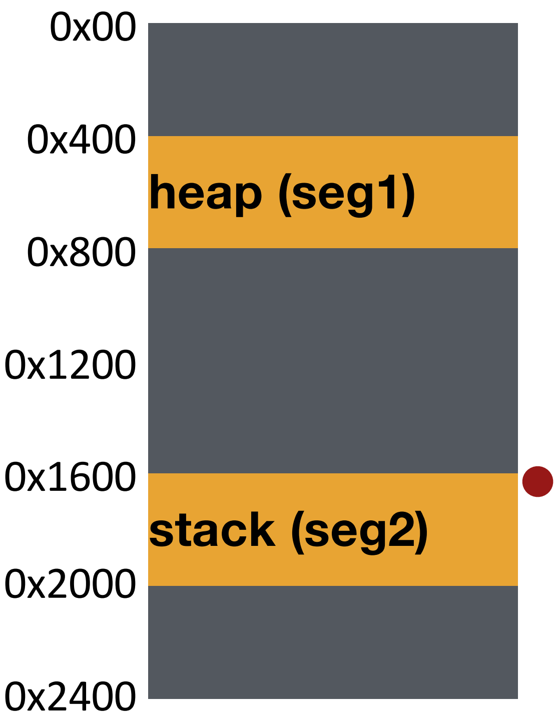
Virtual (hex)

load 0x2010, R1

Physical

Segment numbers:

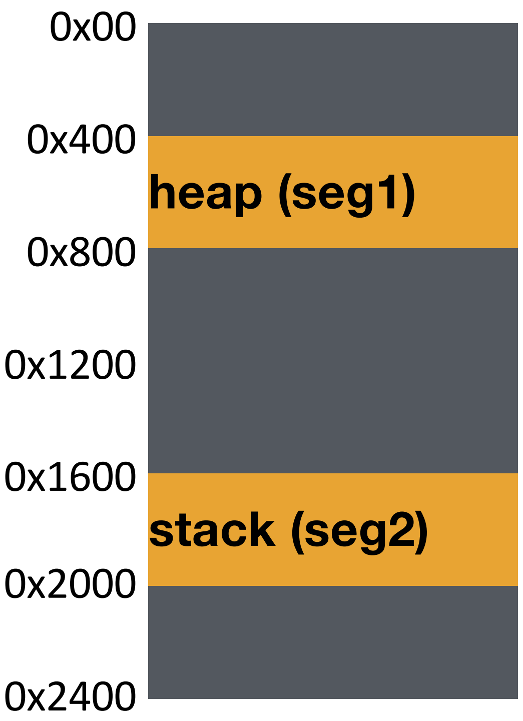
0: code+data
1: heap
2: stack



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

Segment numbers:

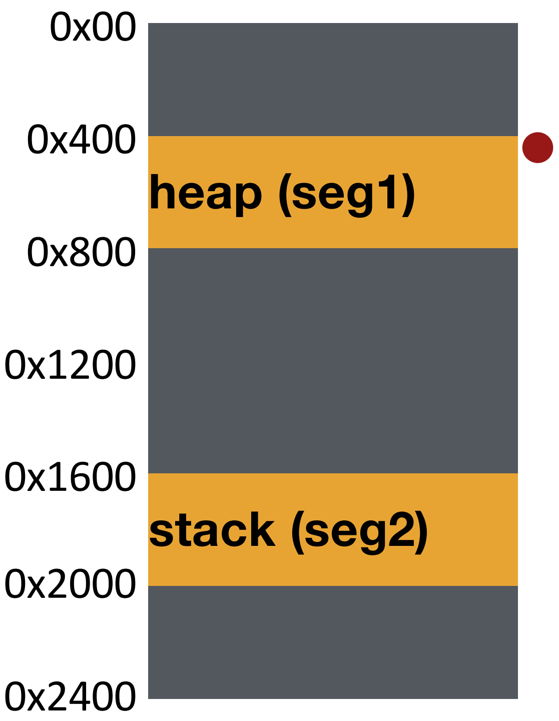
0: code+data
1: heap
2: stack



Virtual (hex)	**Physical**
load 0x2010, R1	0x1600 + 0x010 = 0x1610
load 0x1010, R1	

Segment numbers:

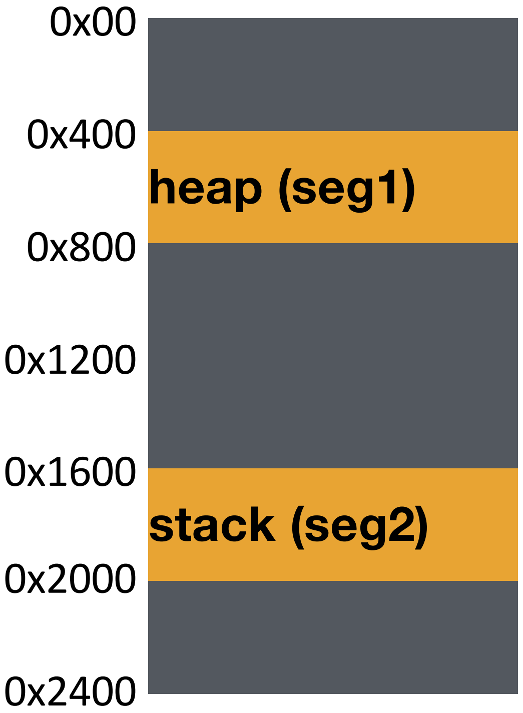
0: code+data
1: heap
2: stack



Virtual (hex)	Physical
load 0x2010, R1	0x1600 + 0x010 = 0x1610
load 0x1010, R1	0x400 + 0x010 = 0x410

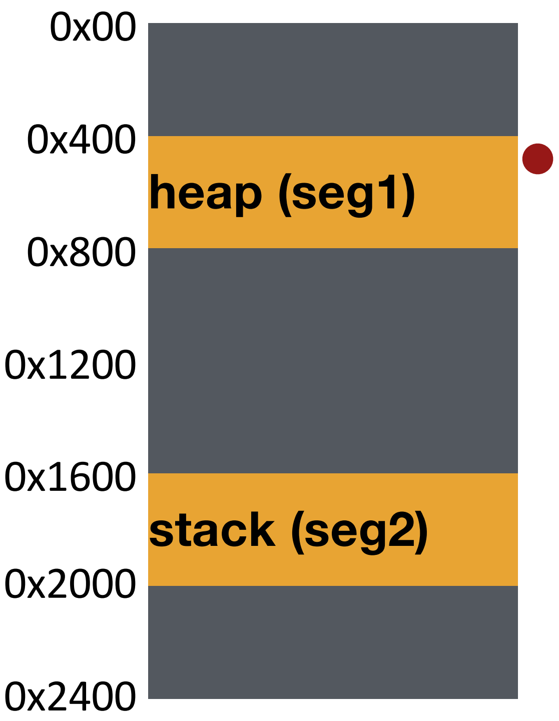
Segment numbers:

0: code+data
1: heap
2: stack



Virtual	Physical
load 0x2010, R1	0x1600 + 0x010 = 0x1610
load 0x1010, R1	0x400 + 0x010 = 0x410
load 0x1100, R1	

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



Virtual	Physical
load 0x2010, R1	0x1600 + 0x010 = 0x1610
load 0x1010, R1	0x400 + 0x010 = 0x410
load 0x1100, R1	0x400 + 0x100 = 0x500

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

Memory accesses every instruction

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

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

Physical Memory Accesses?

1) Fetch instruction at logical addr 0x0010

- Physical addr: 0x4010

Exec, load from logical addr 0x1100

- Physical addr: 0x5900

2) Fetch instruction at logical addr 0x0013

- Physical addr: 0x4013

Exec, no load

3) Fetch instruction at logical addr 0x0019

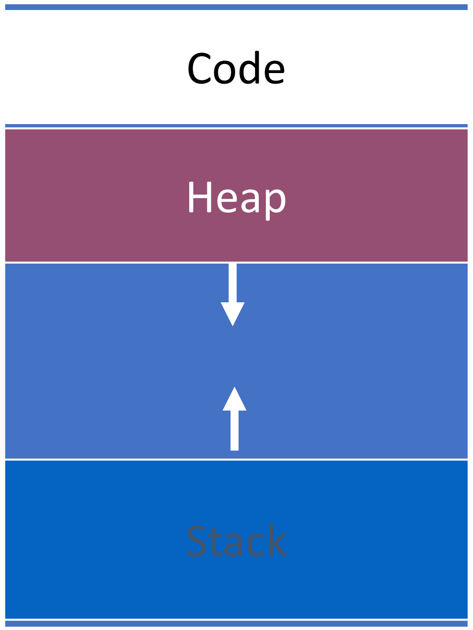
- Physical addr: 0x4019

Exec, store to logical addr 0x1100

- Physical addr: 0x5900

Advantages of Segmentation

- Enables sparser allocation of memory address space than one base+bounds
  - Stack and heap can grow independently
  - Heap: If no data on free list, dynamic memory allocator requests more from OS (e.g., UNIX: malloc calls sbrk())
  - Stack: OS recognizes reference outside legal segment, extends stack implicitly
- Different protection for different segments
  - Read-only status for code
- Enables sharing of some segments as desired
- Supports dynamic relocation of each segment



Disadvantages of Segmentation?

Each segment must be allocated contiguously

- May not have sufficient physical memory for large segments!

- Cannot support holding a part of a large segment in memory

Disadvantages of Segmentation?

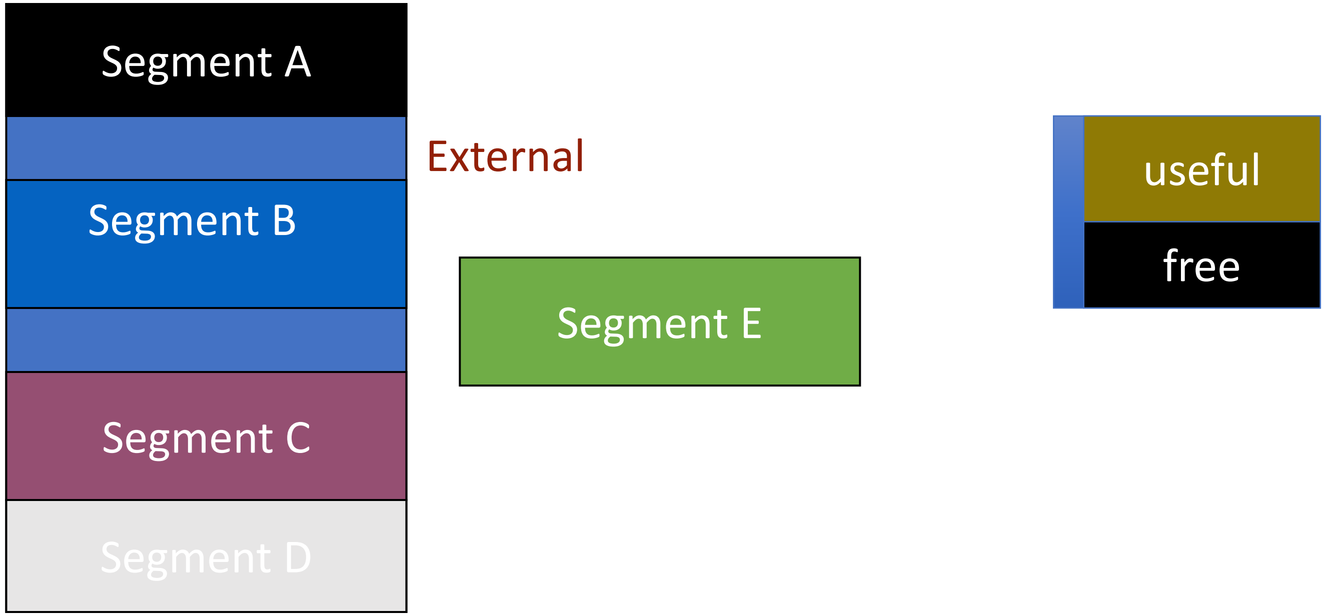
Fragmentation: Free memory that can’t be usefully allocated

Why? Free memory (hole) is too small and scattered

- Segmentation prohibits using this free space since segment is “indivisible”

Types of fragmentation

- External: Visible to allocator (e.g., OS)
- Internal: Visible to requester (e.g., if must allocate at some granularity)



HW+OS work together to virtualize memory

- Give illusion of private address space to each process

Add MMU registers for base+bounds so translation is fast

- OS not involved with every address translation, only on context switch or errors

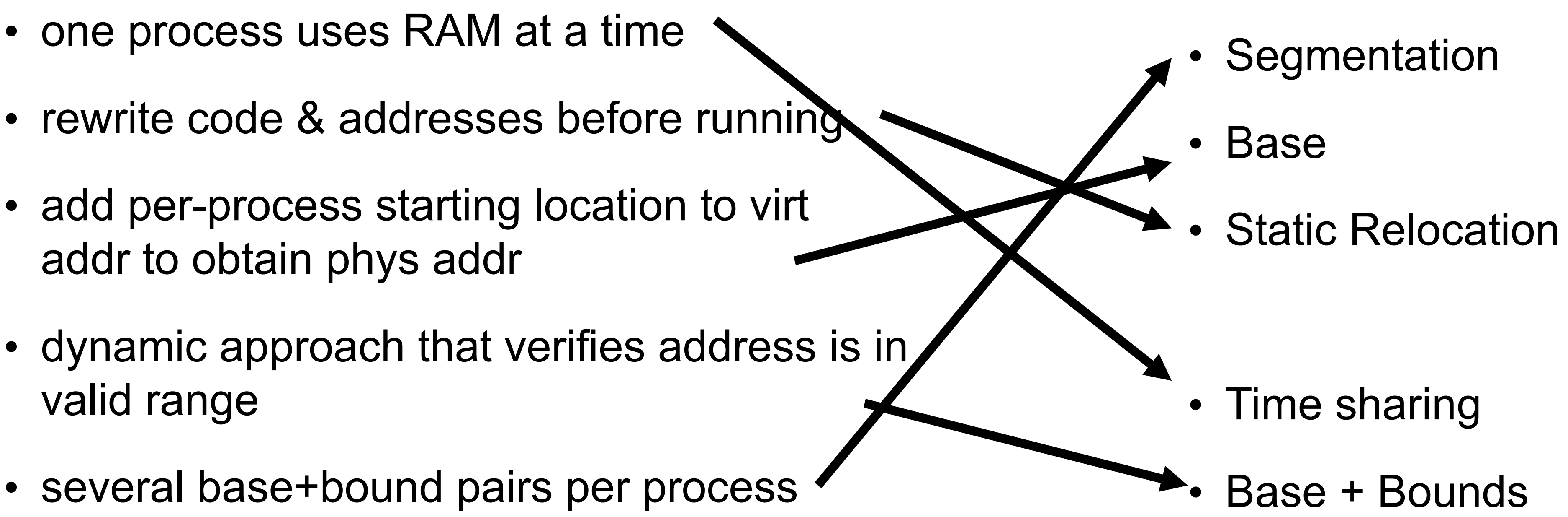
Dynamic relocation with segments is good building block

- Next: Solve fragmentation with paging

Review: Match Description

- Description

- Name of approach
  (covered previous lecture):



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

- Segmentation
- Base
- Static Relocation
- Time sharing
- Base + Bounds

Paging

Questions we answer:

What is paging?

Where are page tables stored?

What are advantages and disadvantages of paging?

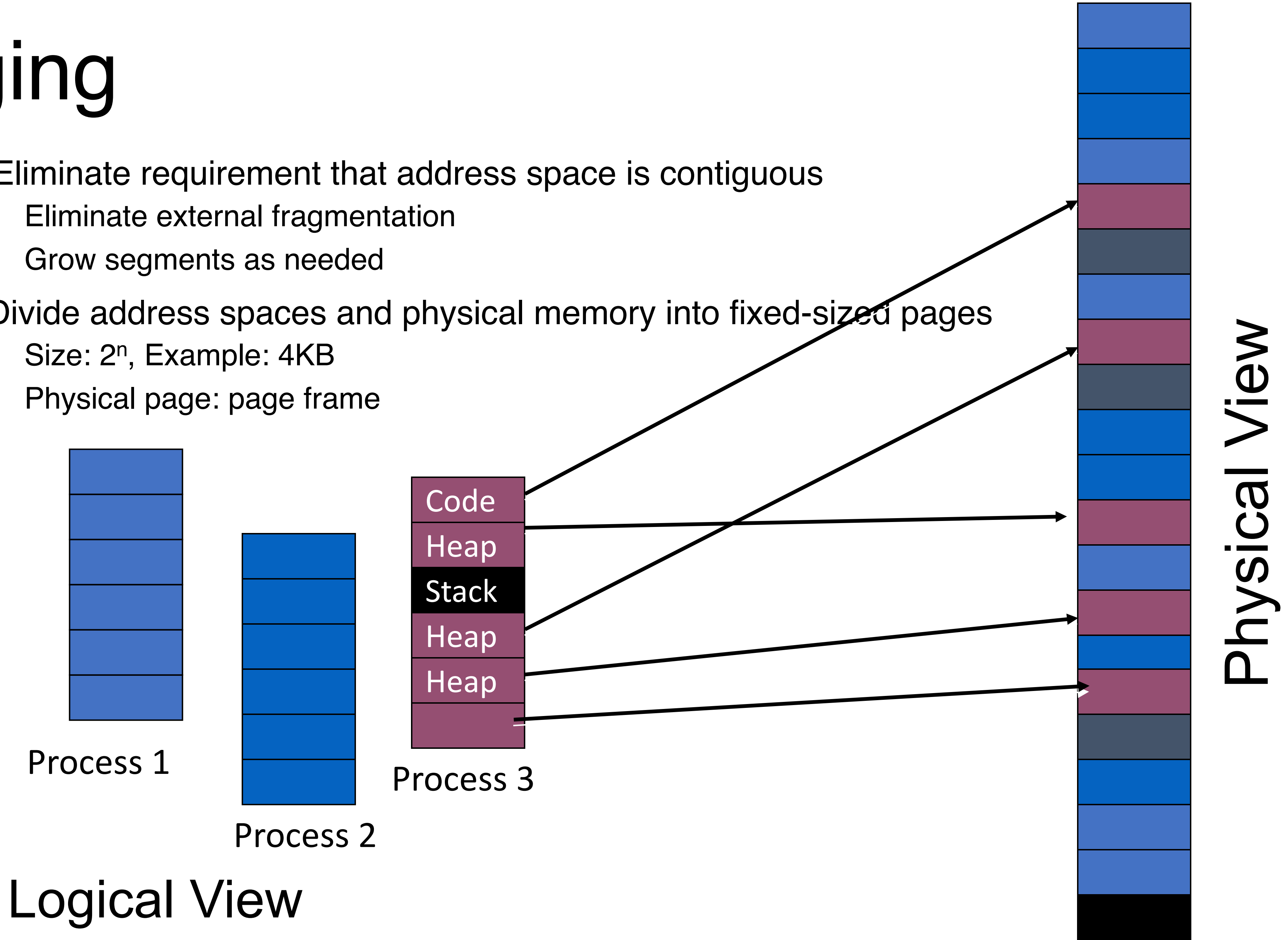
Paging

Goal: Eliminate requirement that address space is contiguous

- Eliminate external fragmentation
- Grow segments as needed

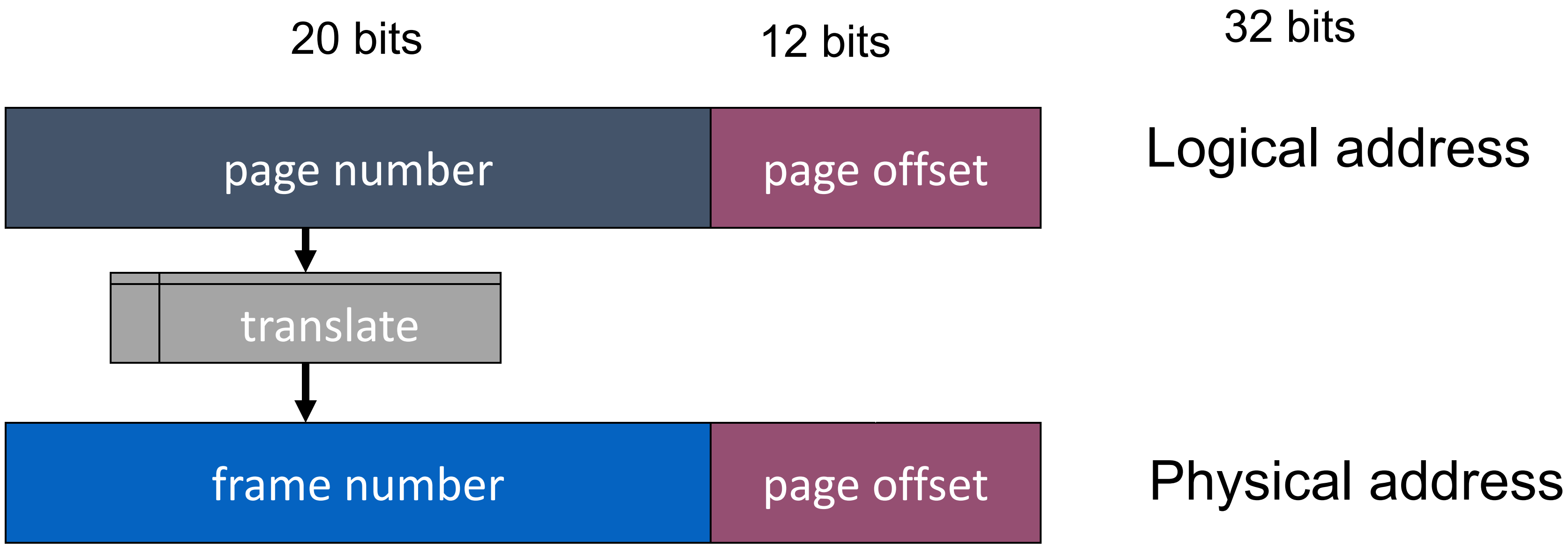
Idea: Divide address spaces and physical memory into fixed-sized pages

- Size: 2^n, Example: 4KB
- Physical page: page frame



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; just append bits correctly…

How does format of address space determine number of pages and size of pages?

Impact of Address Format

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

Page Size	Low Bits (offset)
16 bytes	4
1 KB	10
1 MB	20
512 bytes	9
4 KB	12

Impact of Address Format

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

Page Size	Low Bits (offset)	Virt Addr Bits	High Bits (vpn)
16 bytes	4	10	6
1 KB	10	20	10
1 MB	20	32	12
512 bytes	9	16	7
4 KB	12	32	20

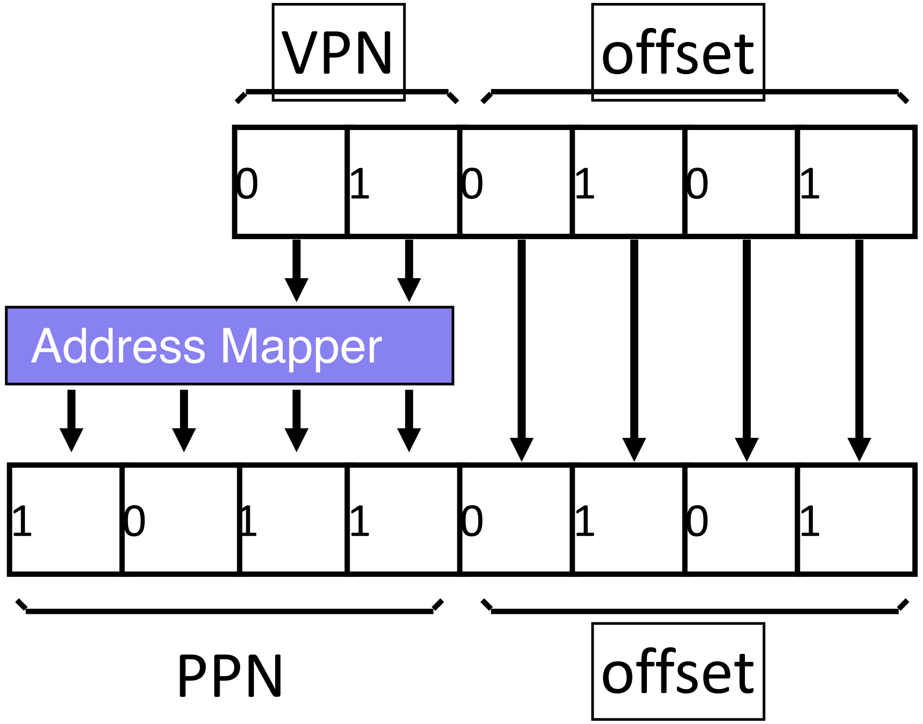
Impact of Address Format

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)	Virt Pages
16 bytes	4	10	6	64
1 KB	10	20	10	1 K
1 MB	20	32	12	4 K
512 bytes	9	16	5	32
4 KB	12	32	20	1 MB

Virtual => Physical PAGE Mapping

Number of bits in virtual address format does not need to equal number of bits in physical address format



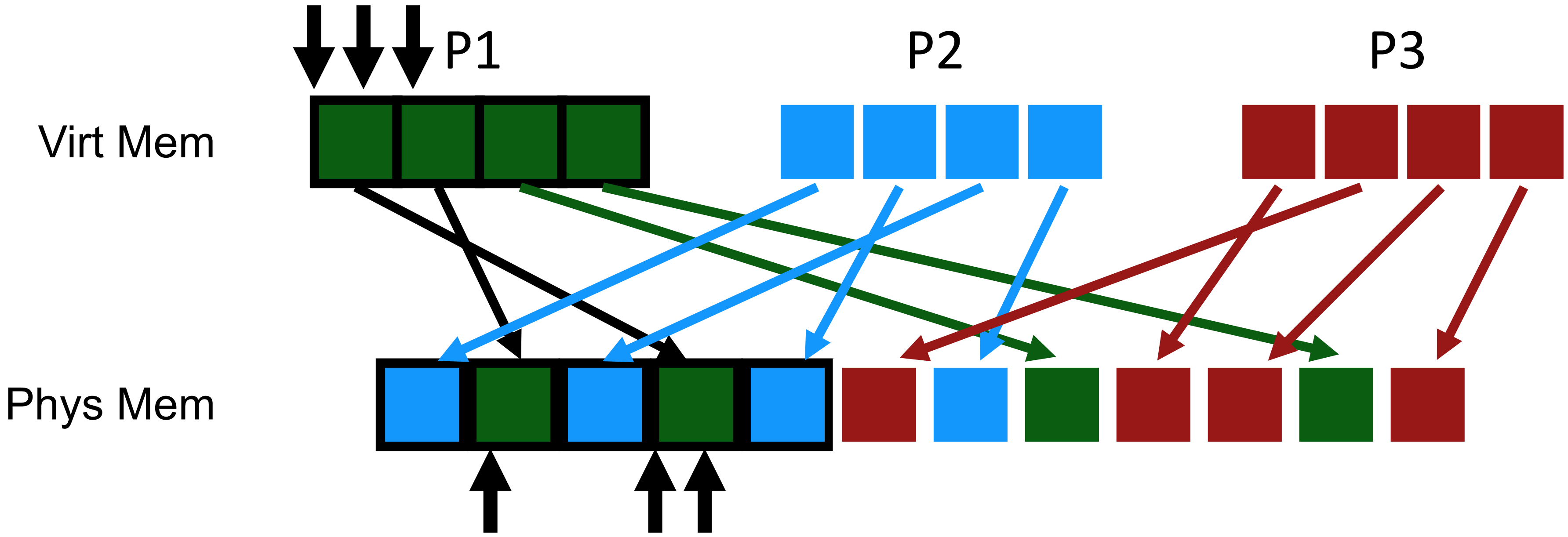
How should OS translate VPN to PPN?

For segmentation, OS used a formula (e.g., phys addr = virt_offset + base_reg)

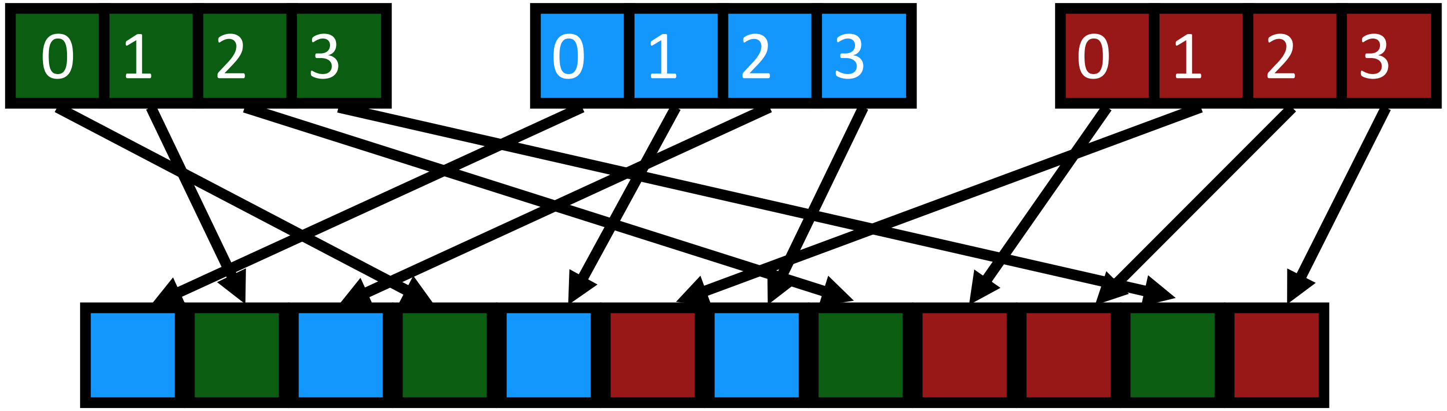
For paging, OS needs more general mapping mechanism

What data structure is good? Big array: page table

The Mapping



Let’s fill in the Page Table



Page Tables:

P1	P2	P3
3	0	8
1	4	5
7	2	9
10	6	11

Where Are Pagetables Stored?

How big is a typical page table?

- assume **32-bit** address space
- assume 4 KB pages
- assume 4 byte entries

- Page table size = Num entries * size of each entry
- Num entries = num virtual pages = 2^(bits for vpn)
- Bits for vpn = 32– number of bits for page offset

  = 32 – lg(4KB) = 32 – 12 = 20
- Num entries = 2^20 = 1 MB
- Page table size = Num entries * 4 bytes = 4 MB

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 PT 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 Pages

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

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

Old: How many mem refs with segmentation?

5 (3 instrs, 2 movl)

Physical Memory Accesses with Paging?

1) Fetch instruction at logical addr 0x0010; vpn?

- 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; vpn?

- 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)

Pagetable is slow!!! Doubles memory references

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 coallesce with adjacent free space
- Just use bitmap to show free/allocated page frames

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

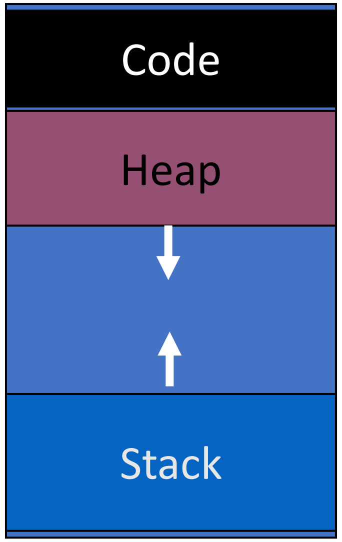
- Wasted memory grows with larger pages
- **Tension**

Additional memory reference to page table --> time-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 → space-inefficient!

- Simple page table: Requires PTE for all pages in address space
  - Entry needed even if page not allocated
- Problematic with dynamic stack and heap within address space
- Page tables must be allocated contiguously in memory
- Solution: Combine paging and segmentation (future lecture)



Reducing Page Table sizes

How big are page Tables?

1. PTE’s are **2 bytes**, and **32** possible virtual page numbers

   32 * 2 bytes = 64 bytes

2. PTE’s are **2 bytes**, virtual addrs are **24 bits**, pages are **16 bytes**

   2 bytes * 2^(24 – lg 16) = **2^21 bytes** (2 MB)

3. PTE’s are **4 bytes**, virtual addrs are **32 bits**, and pages are **4 KB**

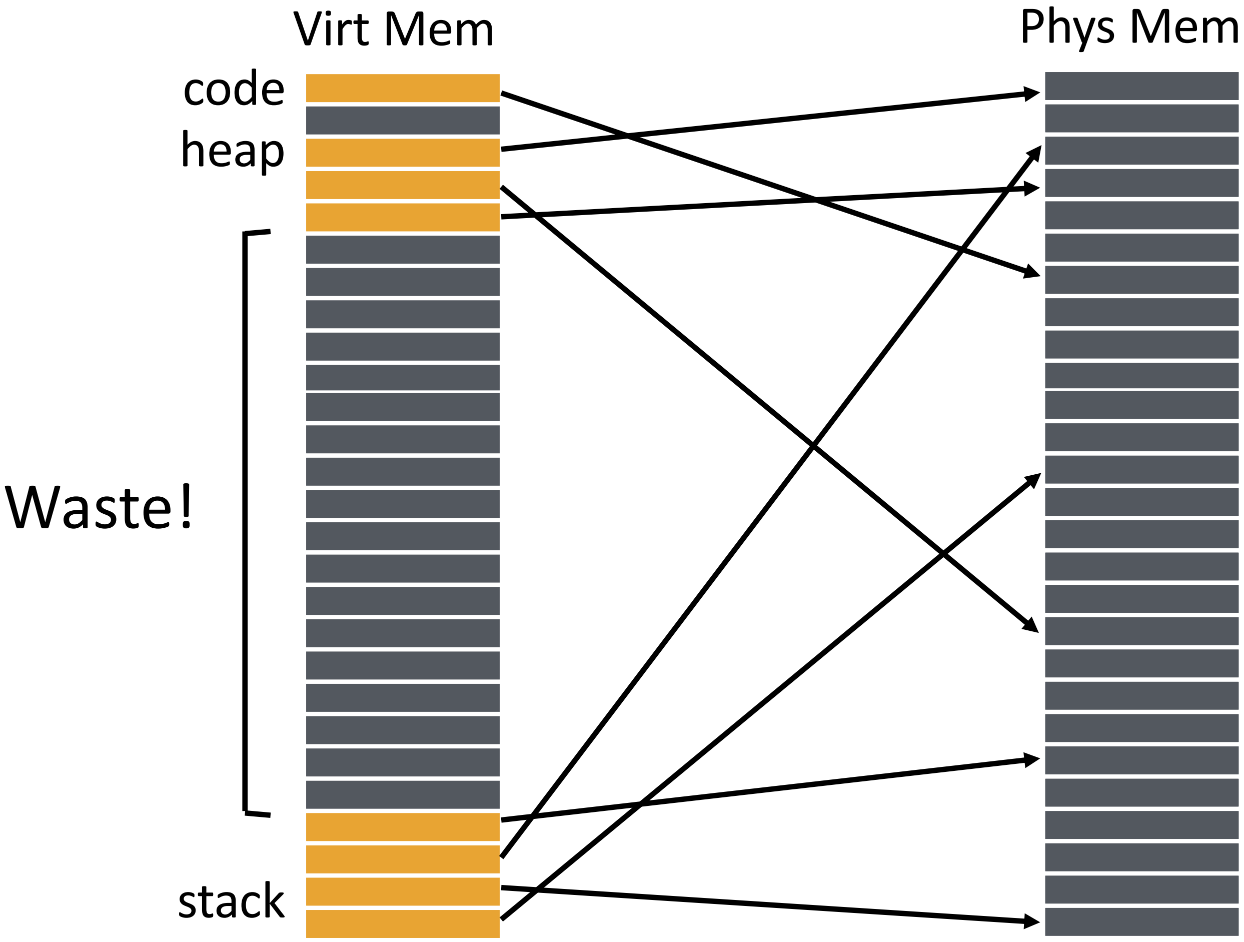
   4 bytes * 2^(32 – lg 4K) = **2^22 bytes** (2 MB)

4. PTE’s are **4 bytes**, virtual addrs are **64 bits**, and pages are **4 KB**

   4 bytes * 2^(64 – lg 4K) = **2^54 bytes**

How big is each page table?

Why ARE Page Tables so Large?



Many invalid PT entries

Format of linear page tables:

PFN	valid	prot
10	1	r-x
-		0 -
-		0 -
-		0 -
-		0 -
-		0 -
…many more invalid…		
-		0 -
-		0 -
-		0 -
-		0 -
28	1	rw-
4	1	rw-
23	1	rw-

how to avoid storing these?

Avoid Simple linear Page Table

Use more complex page tables, instead of just big array

Any data structure is possible (with software-managed TLB)

- Hardware looks for vpn in TLB on every memory access
- If TLB does not contain vpn, TLB miss
  - Trap into OS and let OS find vpn->ppn translation
  - OS notifies TLB of vpn->ppn for future accesses

Approach 1: Inverted Page Table

Inverted Page Tables

- Only need entries for virtual pages w/ valid physical mappings

Naïve approach:
Search through data structure <ppn, vpn+asid> to find match

- Too much time to search entire table

Better: Find possible matches entries by hashing vpn+asid

- Smaller number of entries to search for exact match

Managing inverted page table requires software-controlled TLB

For hardware-controlled TLB, need well-defined, simple approach

Other Approaches

1. Inverted Pagetables
2. Segmented Pagetables
3. Multi-level Pagetables
    - Page the page tables
    - Page the pagetables of page tables…

Valid PTEs are Contiguous

PFN	valid	prot
10	1	r-x
-		0
		-
-		0
-		0
-		0
-		0
…many more invalid…		-
-		0
-		0
-		0
-		0
28	1	rw-
4	1	rw-
23	1	rw-

how to avoid storing these?

Note “hole” in addr space:
valids vs. invalids are clustered

How did OS avoid allocating holes in phys memory?

Segmentation

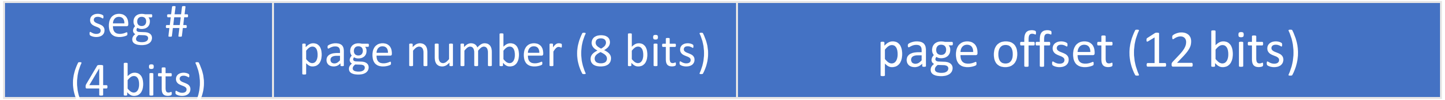
Combine Paging and Segmentation

Divide address space into segments (code, heap, stack)

- Segments can be variable length

Divide each segment into fixed-sized pages

Logical address divided into three portions



seg # (4 bits)

page number (8 bits)

page offset (12 bits)

Implementation

- Each segment has a page table
- Each segment track base (physical address) and bounds of **page table** for that segment

Combining Paging and Segmentation

seg # (4 bits)	page number (8 bits)	page offset (12 bits)

seg	base	bounds	R W
0	0x002000	0xff (255)	1 0
1	0x000000	0x00	0 0
2	0x001000	0x0f (15)	1 1

...	
0x01f	0x001000
0x011	
0x003	
0x02a	
0x013	
...	
0x00c	0x002000
0x007	
0x004	
0x00b	
0x006	
...	

0x002070 read:	0x004070
0x202016 read:	0x003016
0x104c84 read:	error
0x010424 write:	error
0x210014 write:	error
0x203568 read:	0x02a568

Advantages of Paging with Segmentation

Advantages of Segments

- Supports sparse address spaces
  - Decreases size of page tables
  - If segment not used, not needed for page table

Advantages of Pages

- No external fragmentation
- Segments can grow without any reshuffling
- Can run process when some pages are swapped to disk (next lecture)

Advantages of Both

- Increases flexibility of sharing
  - Share either single page or entire segment
  - How?

Disadvantages of Paging with Segmentation

Potentially large page tables (for each segment)

- Must allocate each page table contiguously
- More problematic with more address bits
- Page table size?
    - Assume 2 bits for segment, 18 bits for page number, 12 bits for offset

Each page table is:

= Number of entries * size of each entry

= Number of pages * 4 bytes

= 2^18 * 4 bytes = 2^20 bytes = 1 MB!!!

Other Approaches

1. Inverted Pagetables
2. Segmented Pagetables
3. Multi-level Pagetables
    - Page the page tables
    - Page the pages of page tables…