Memory Virtualization

RUTGERS
UNIVERSITY | NEW BRUNSWICK

3) Dynamic Relocation

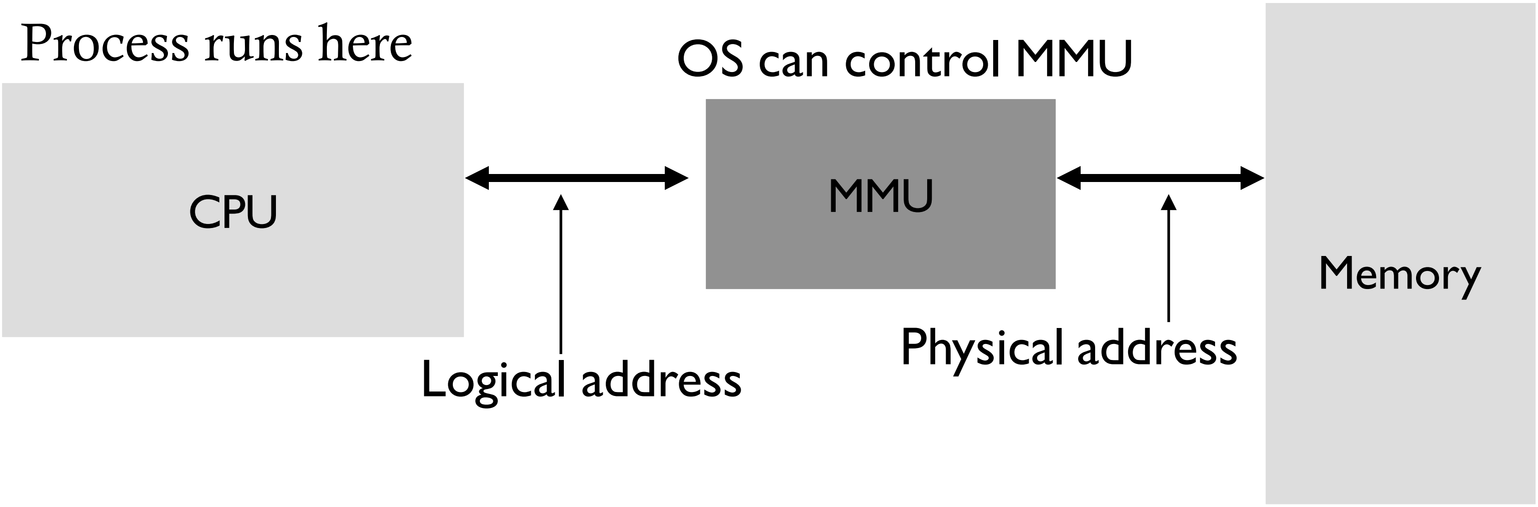
Goal: Protect processes from one another

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**

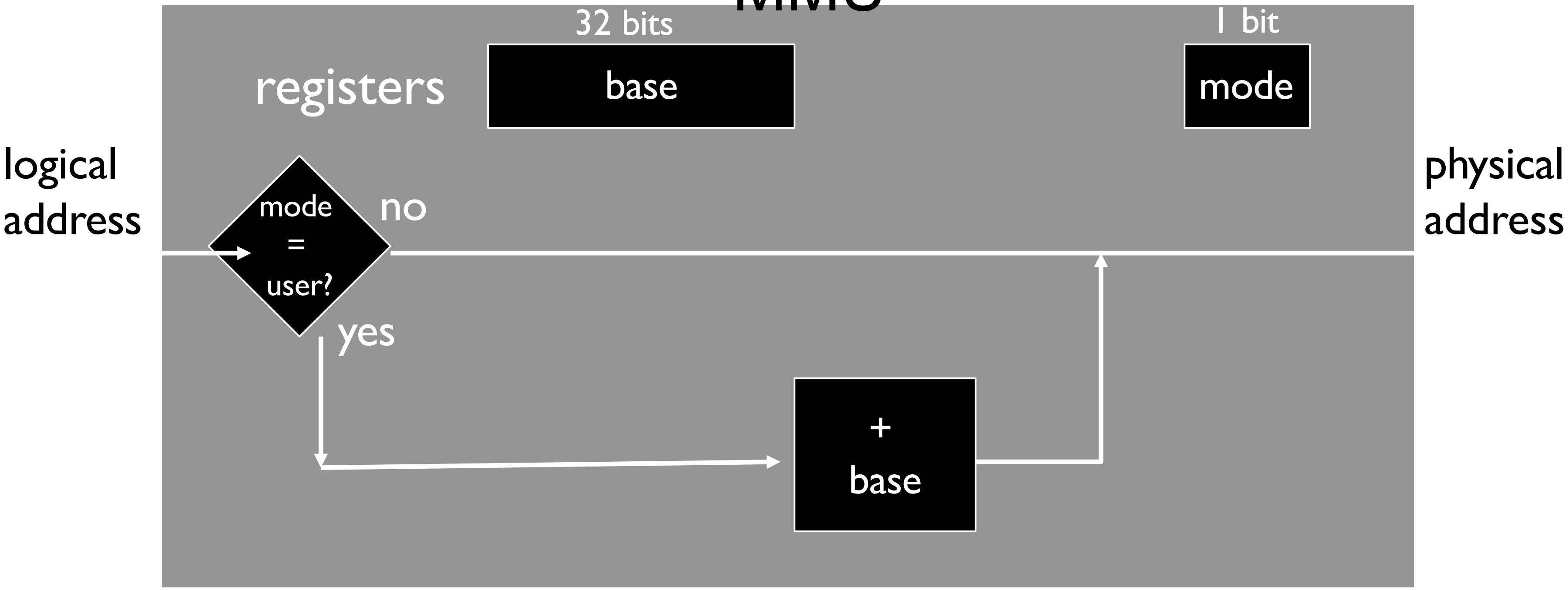
A minimal MMU contains **base register** for translation

- 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

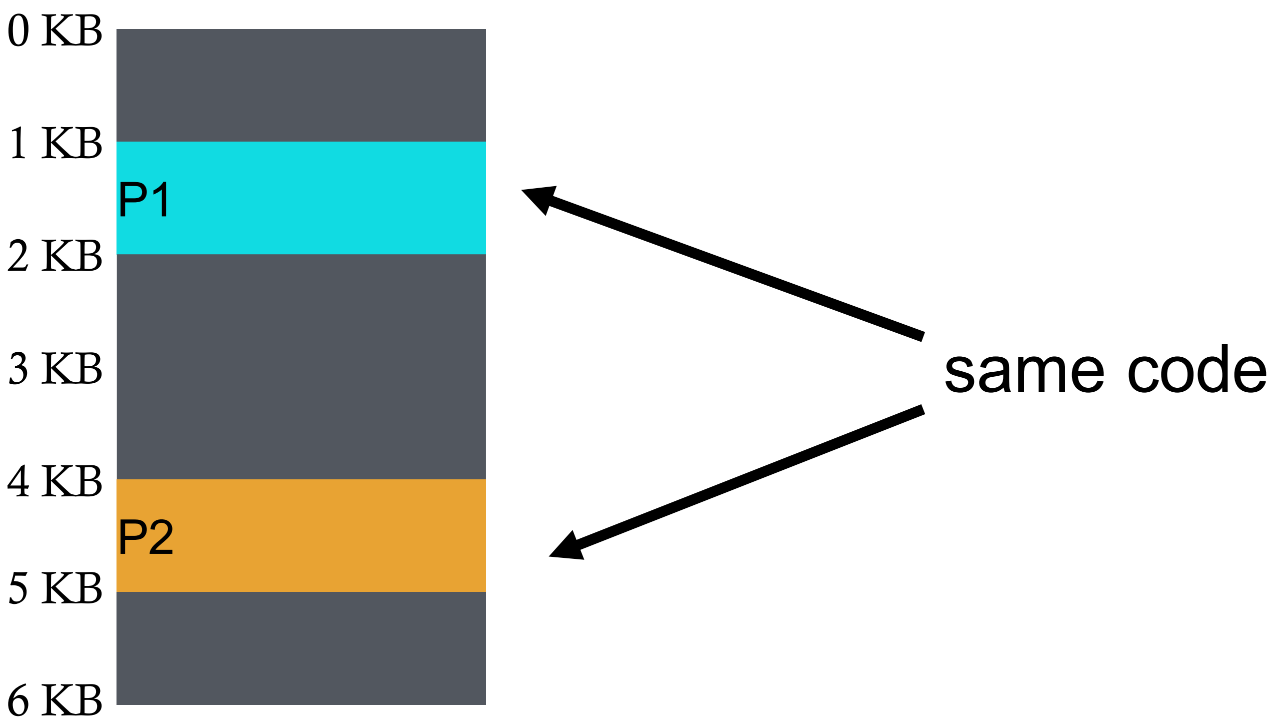


Dynamic Relocation with Base Register

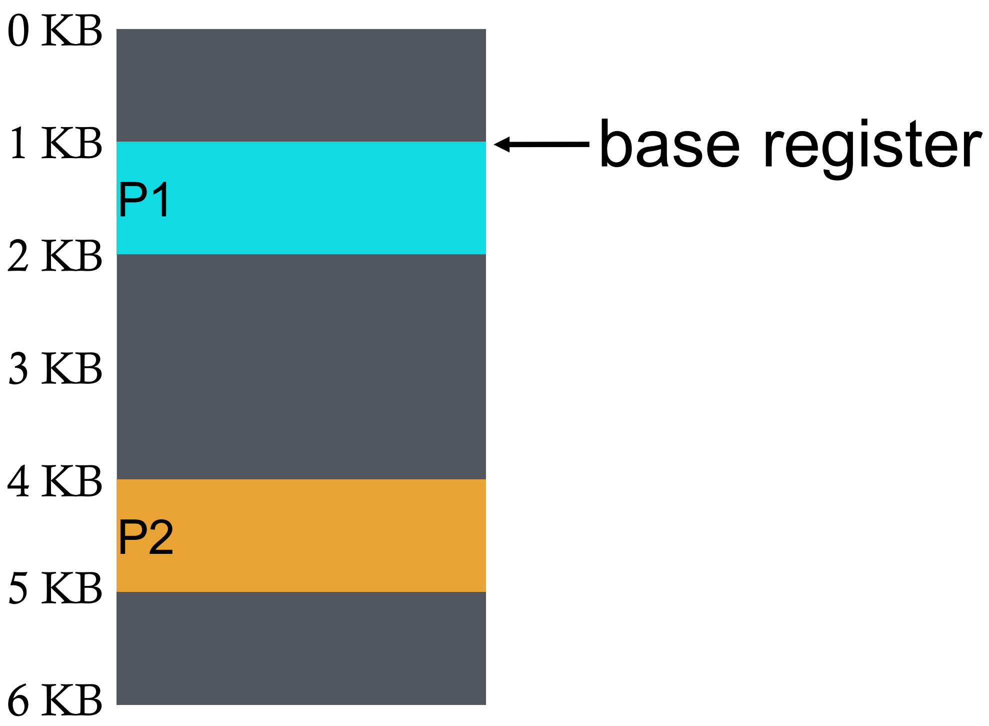
Idea: translate virtual addresses to physical by adding a fixed offset each time.

Store offset in base register

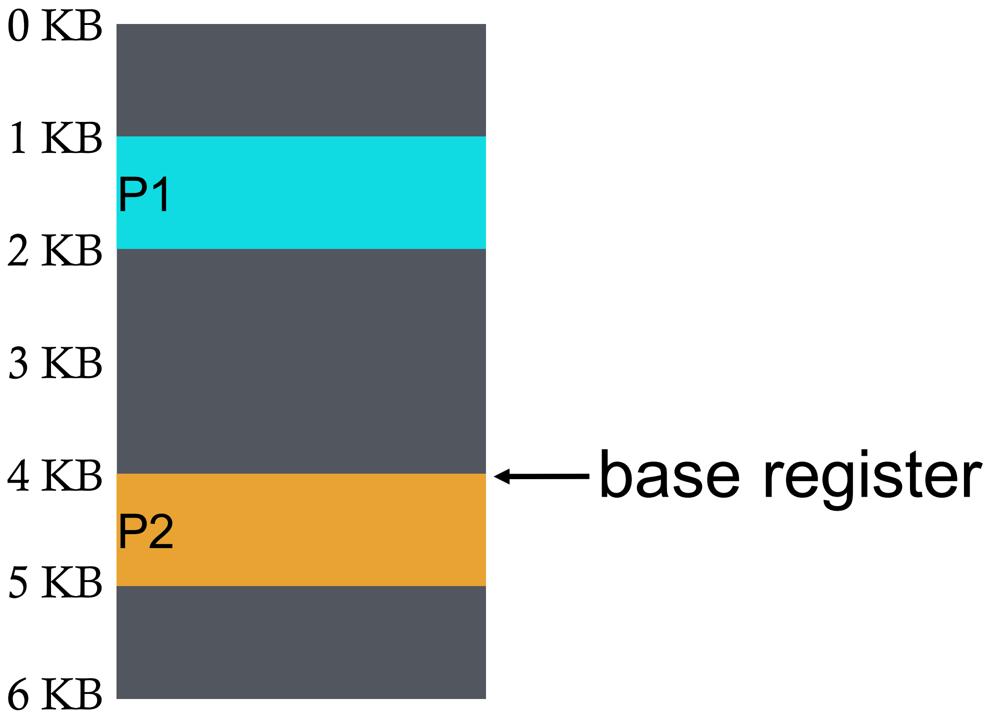
Each process has different value in base register



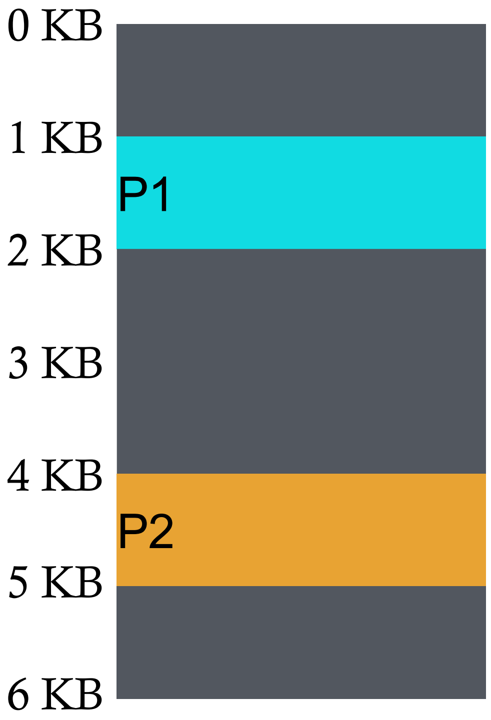
VISUAL Example of DYNAMIC RELOCATION:
BASE REGISTER



P1 is running



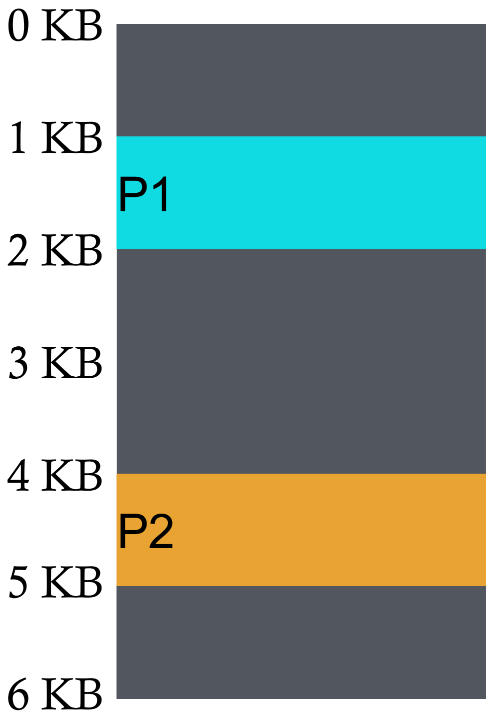
P2 is running



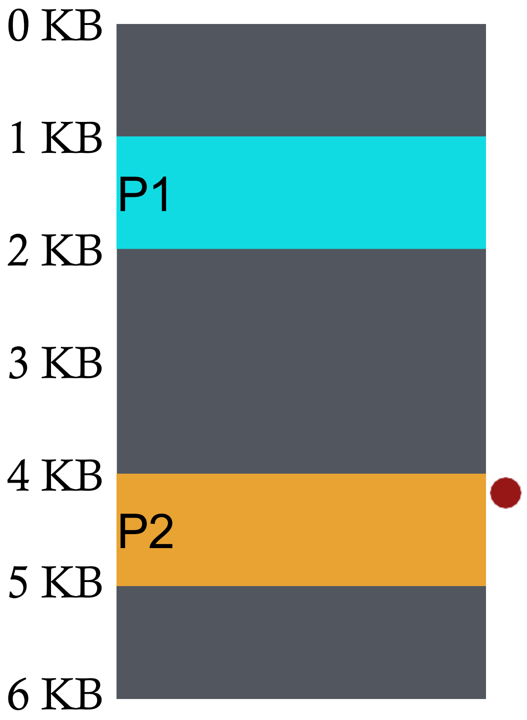
(Decimal notation)

Virtual	Physical
P1: load 100, R1	

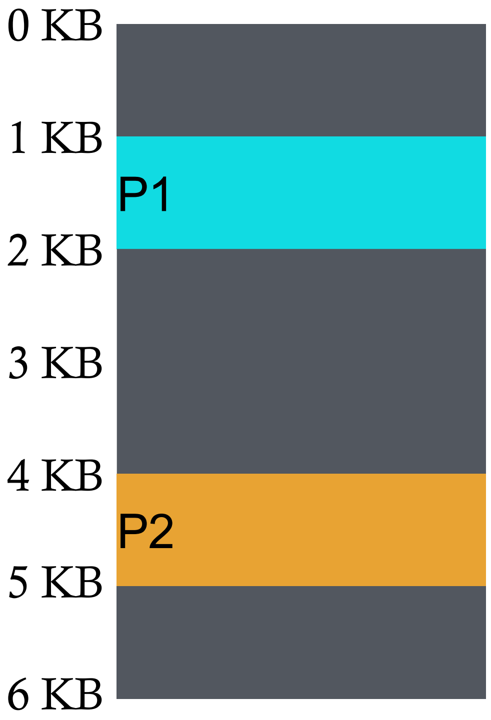
Virtual	Physical	
P1: load 100, R1	load 1124, R1	(1024 + 100)



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

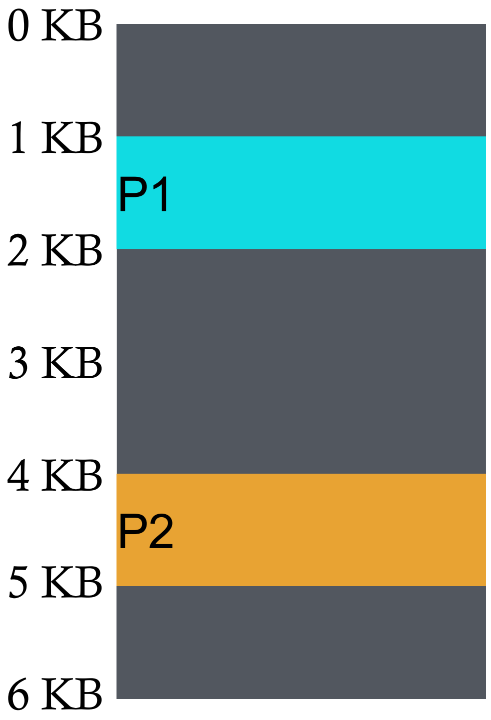


Virtual	Physical	
P1: load 100, R1	load 1124, R1	
P2: load 100, R1	load 4196, R1	(4096 + 100)



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

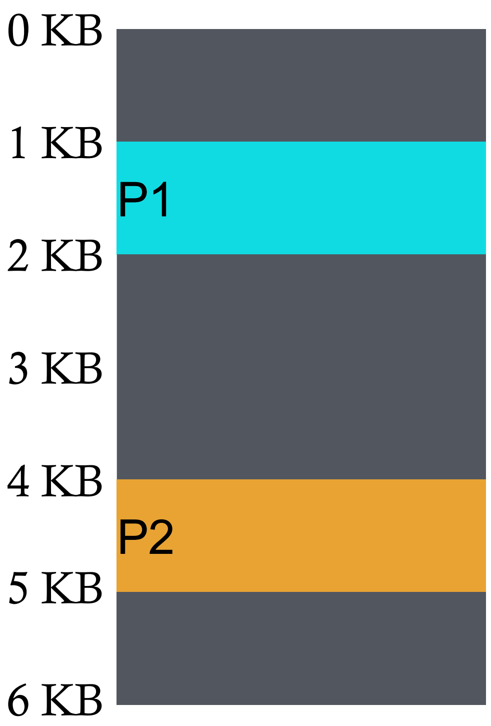
Who Controls the Base Register?

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?

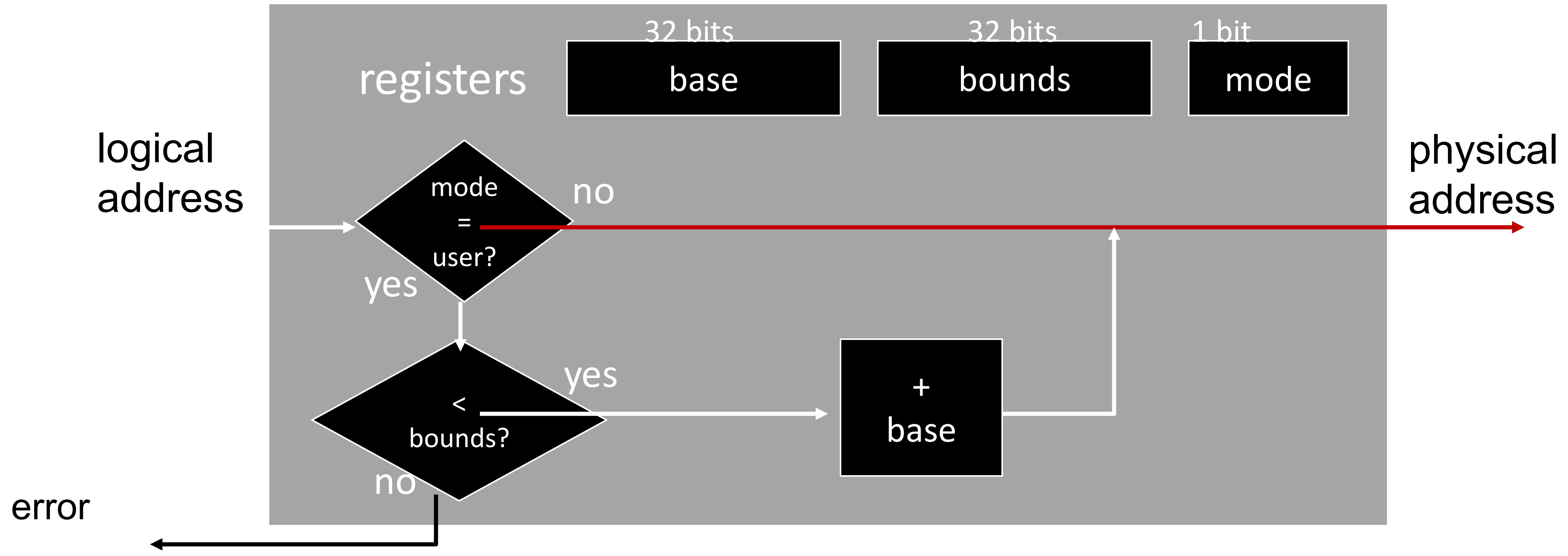
4) Dynamic with Base+Bounds

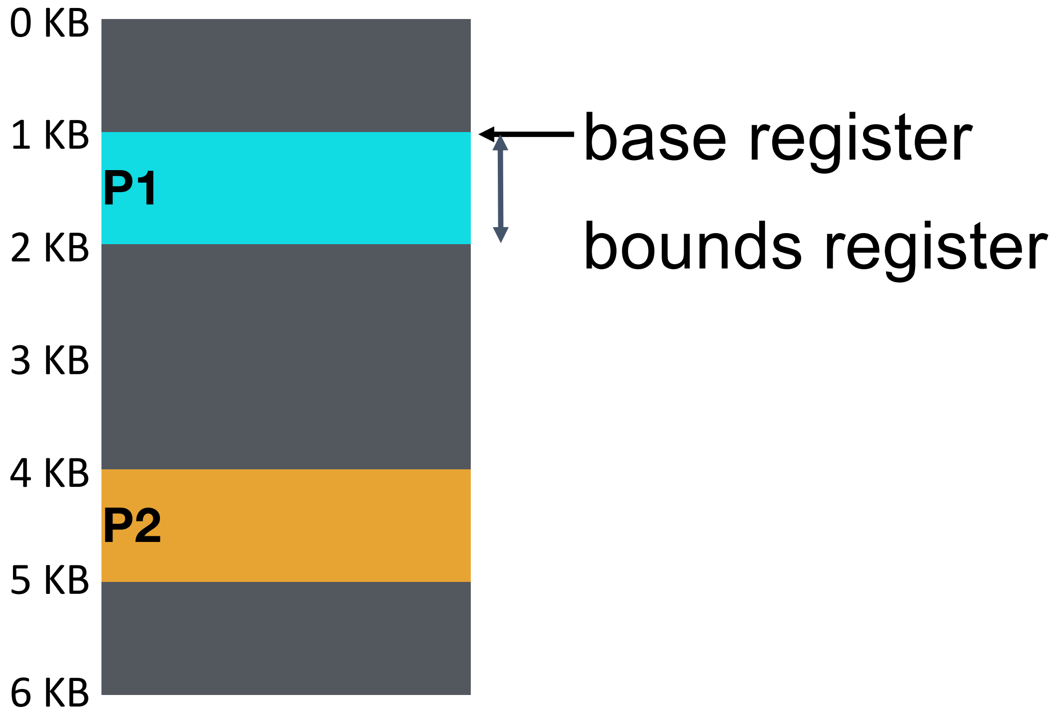
- Idea: limit the address space with a bounds register
- Base register: smallest physical addr (or starting location)
- Bounds register: size of this process's virtual address space
  - Sometimes defined as largest physical address (base + size)
- OS kills process if process loads/stores beyond bounds

Implementation of BASE+BOUNDS

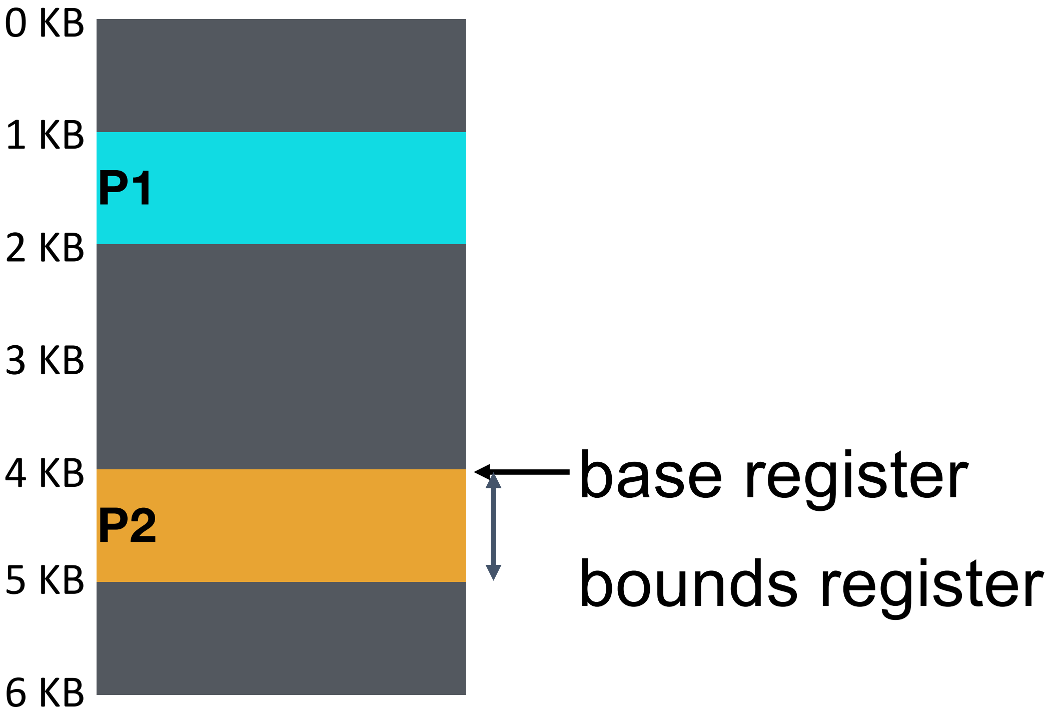
Translation on every memory access of user process

- 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

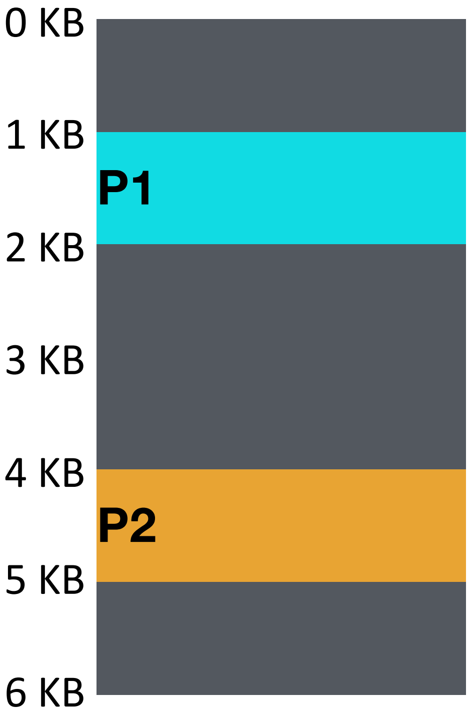




P1 is running

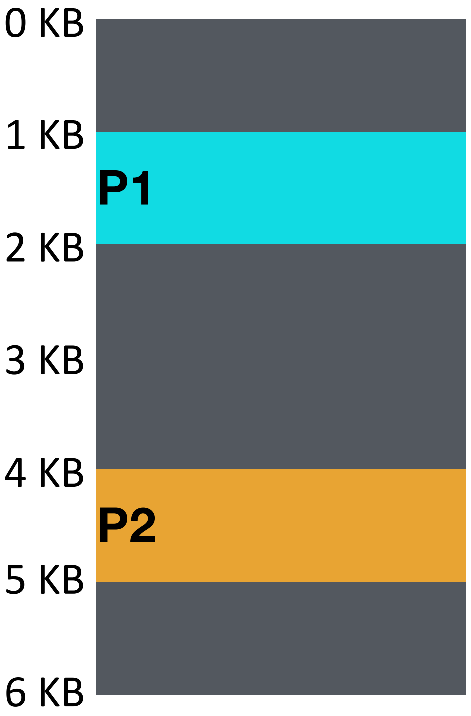


P2 is running



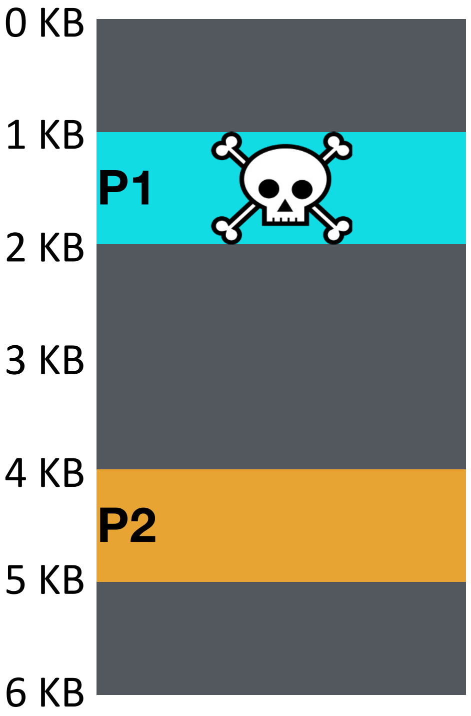
Virtual	Physical
P1: load 100, R1	load 1124, R1
P2: load 100, R1	load 4196, R1
P2: load 1000, R1	load 5196, R1
P1: load 1000, R1	load 2024, R1
P1: store 3072, R1	

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

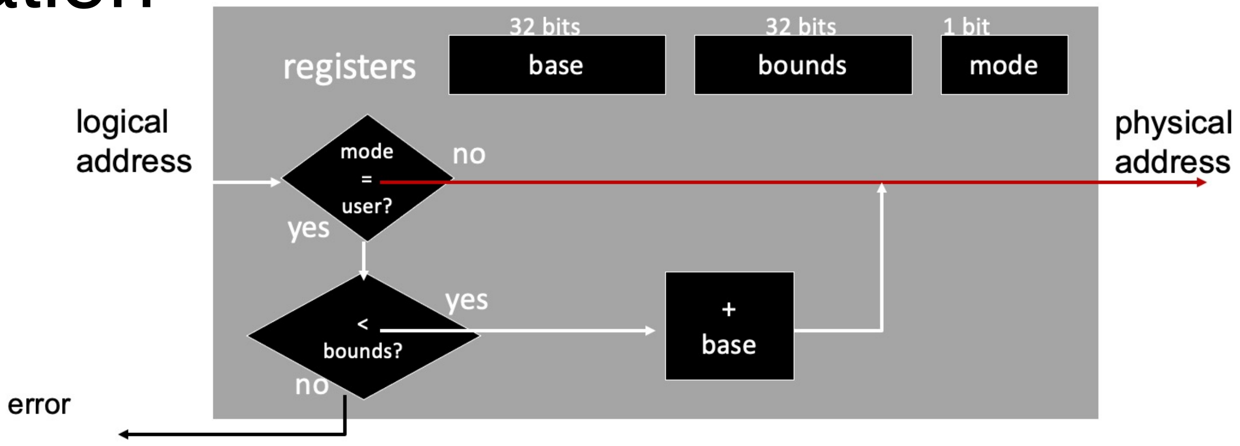
- Add base and bounds registers to Process Control Block
- Steps
  - Change to privileged mode
  - Save base and bounds registers of old process
  - Load base and bounds registers of new process
  - Change to user mode and jump to new process

Protection requirements

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

Base and Bounds Advantages

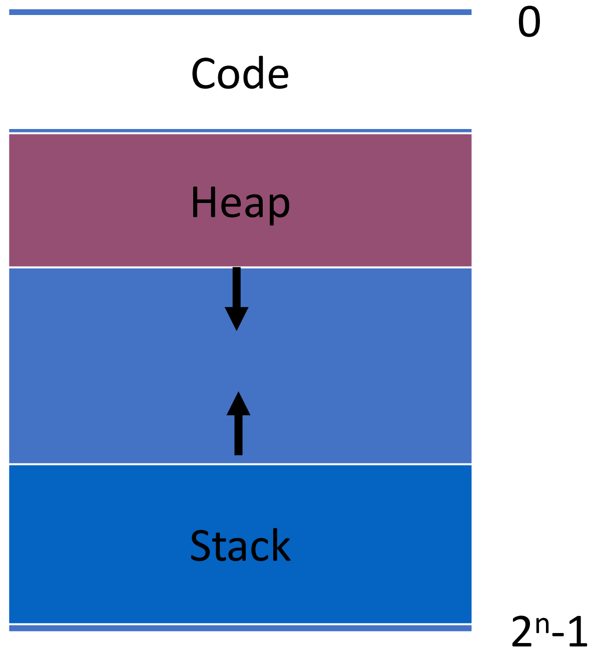
- Provides protection (both read and write) across address spaces
- Supports dynamic relocation
  - Can place process initially at locations different from assumed in the program code
  - Also, move address spaces later if needed
- Simple, inexpensive implementation
  - Few registers, little logic in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



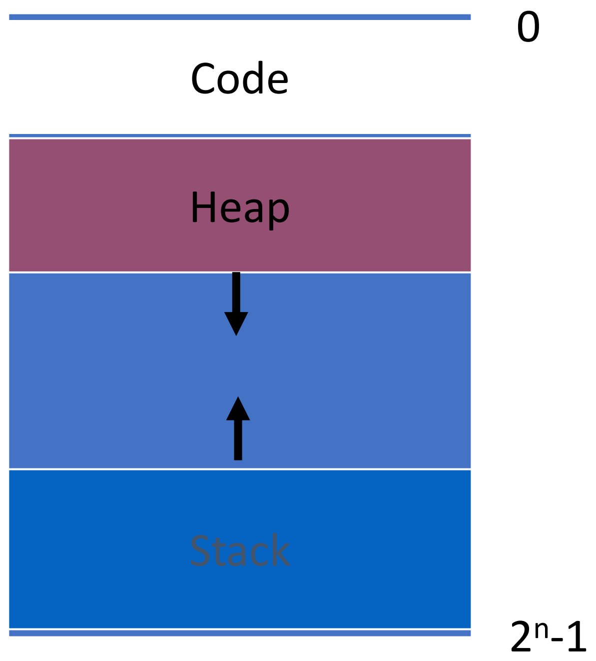
5) Segmentation

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

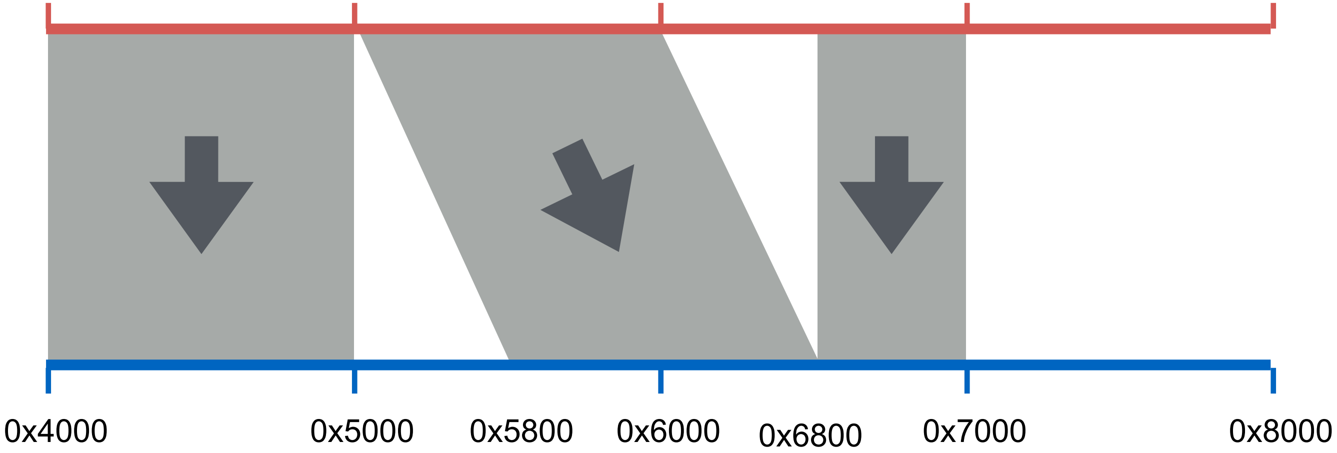
0x0240:

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) | **Physical**

load 0x2010, R1

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

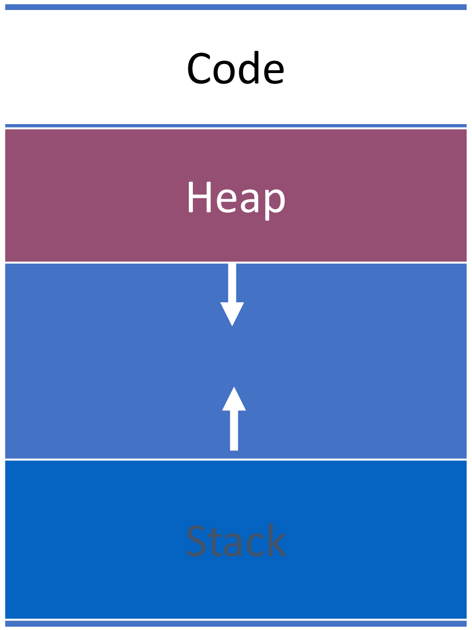
Exec, no load

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?

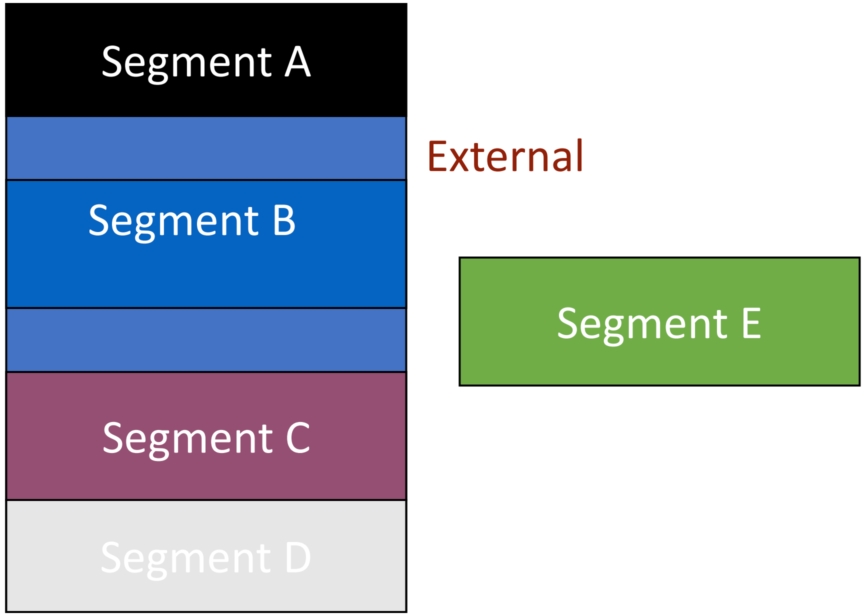
Fragmentation: Free memory that can't be usefully allocated

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

- 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

- Name of approach (covered previous lecture):

- 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?

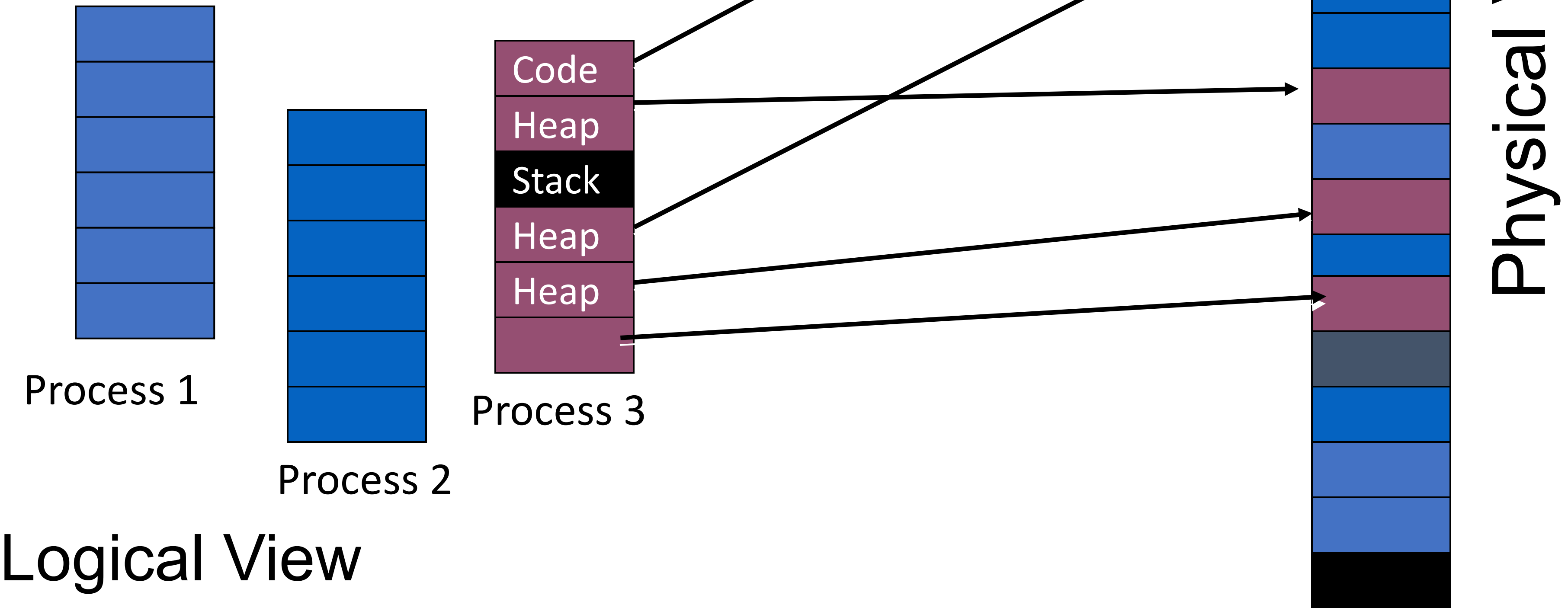
Paging

Goal: Eliminate requirement that address space is contiguous

- Eliminate external fragmentation
- Grow segments as needed

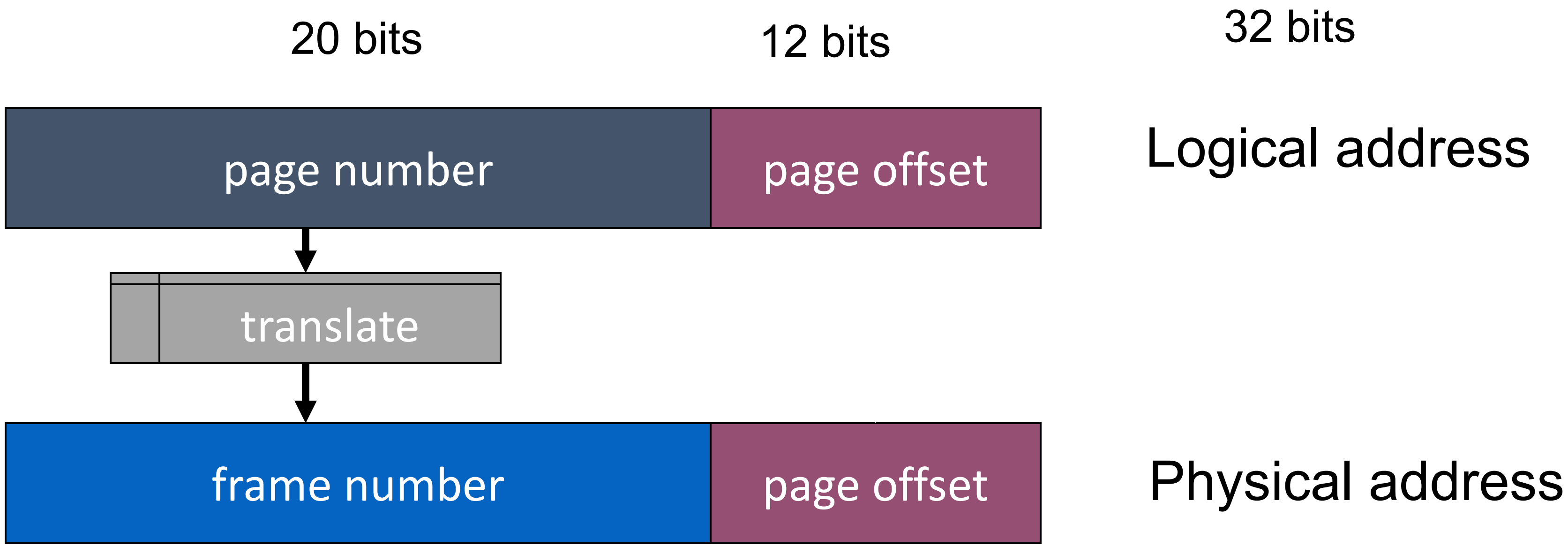
Idea: Divide address spaces and physical memory into fixed-sized pages

- 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