Lecture 7: Dynamic Relocation and Segmentation

Operating Systems

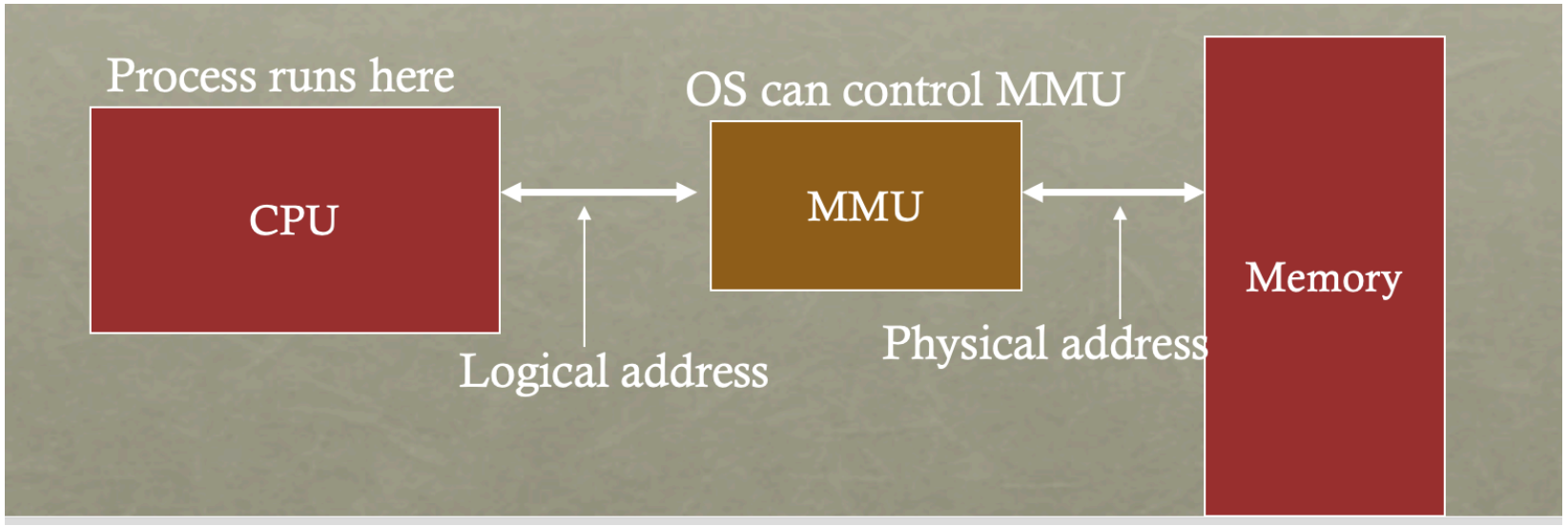
Content taken from: https://www.cse.iitb.ac.in/~mythili/os/

https://pages.cs.wisc.edu/~remzi/OSTEP/

Last Class

- **Virtual Address Space:** Every process assumes it has access to a large space of memory from address 0 to a MAX
- CPU issues loads and stores to virtual addresses
- Address translation is required to go from virtual addresses (VA) to physical addresses (PA)
- **Memory Management Unit (MMU):** Hardware which performs the address translation from VA to PA

Address Translation using MMU



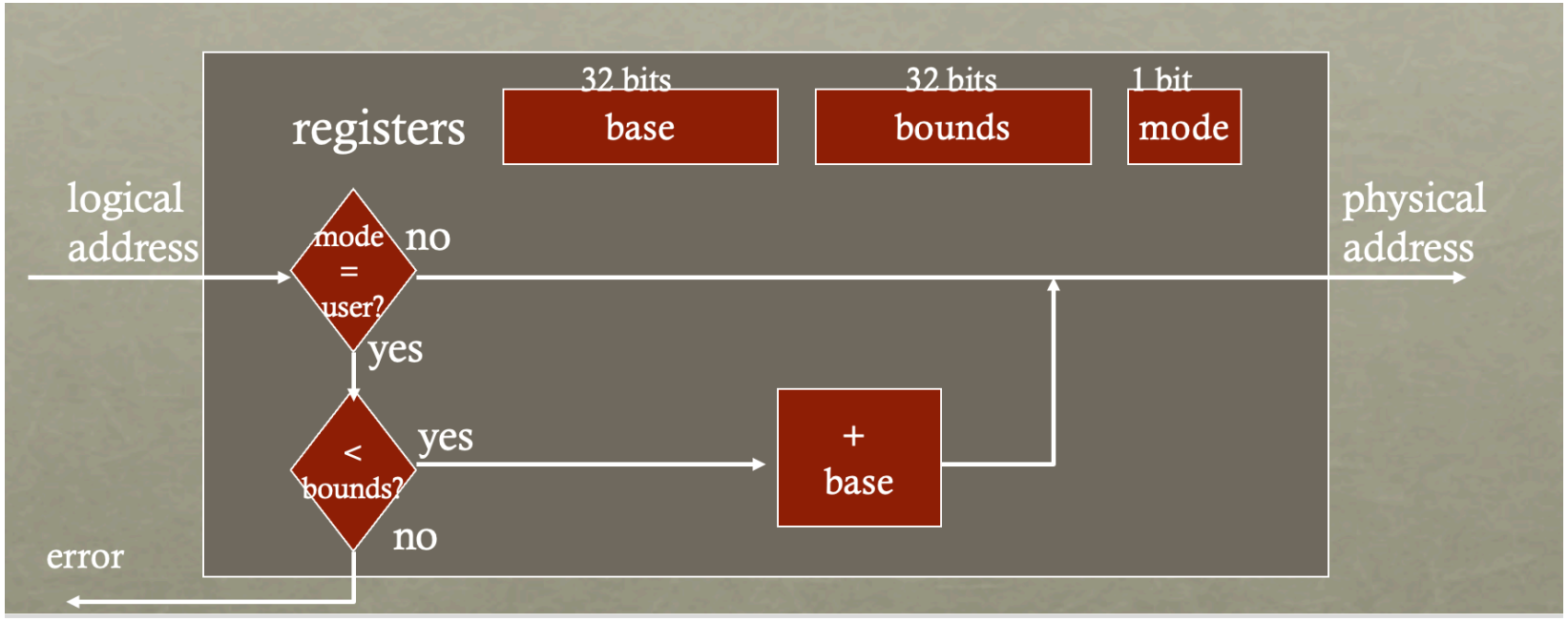
Dynamic Relocation with Base and Bounds

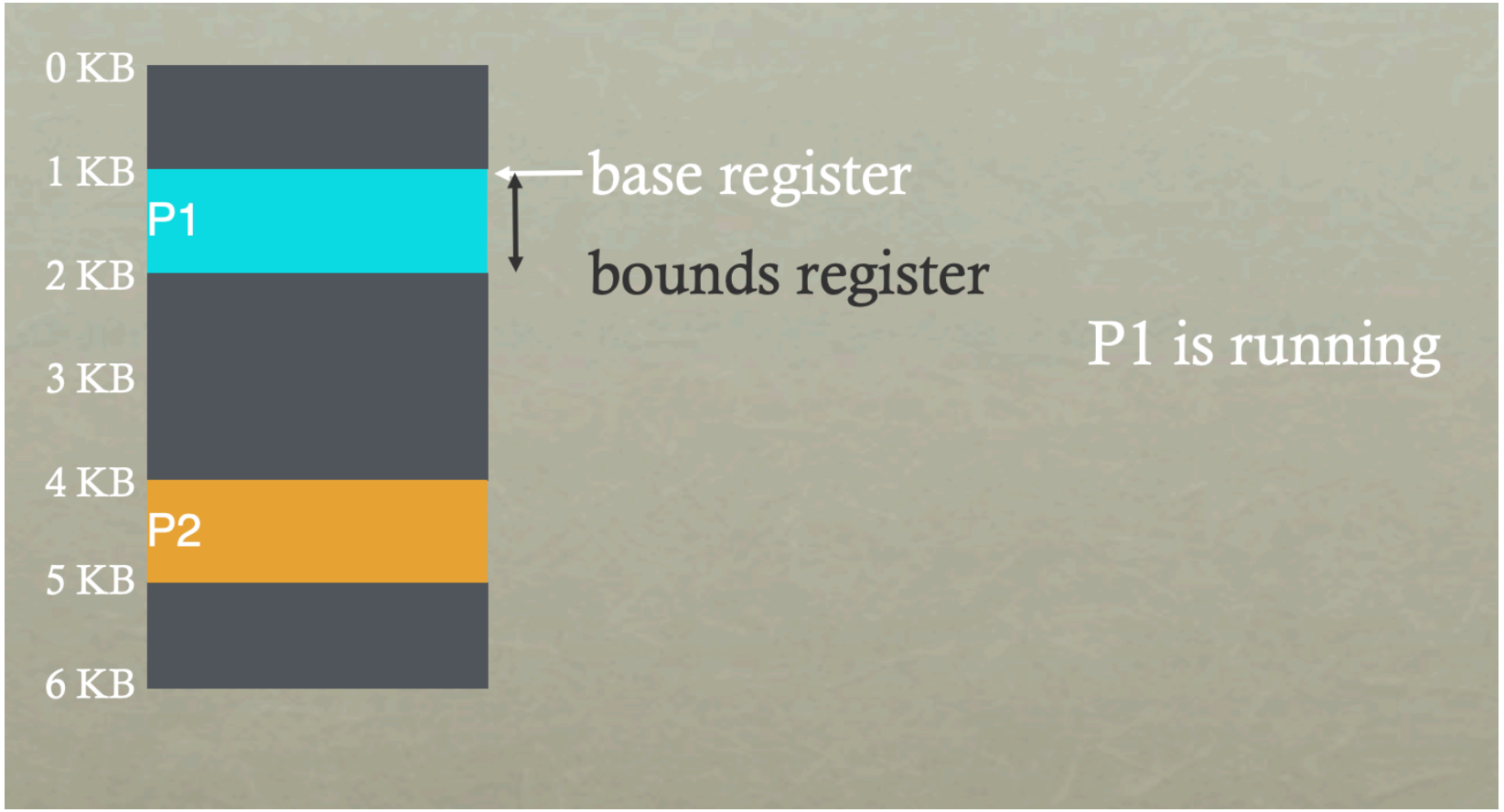
- **Base register:** Contains the offset to be added to VA to get the PA
  - smallest physical address (or starting location)
- **Bounds register:** Ensures that all virtual addresses generated by the process are legal and within the “bounds” of the process.
  - size of this process’s virtual address space

Implementation of Base + Bounds

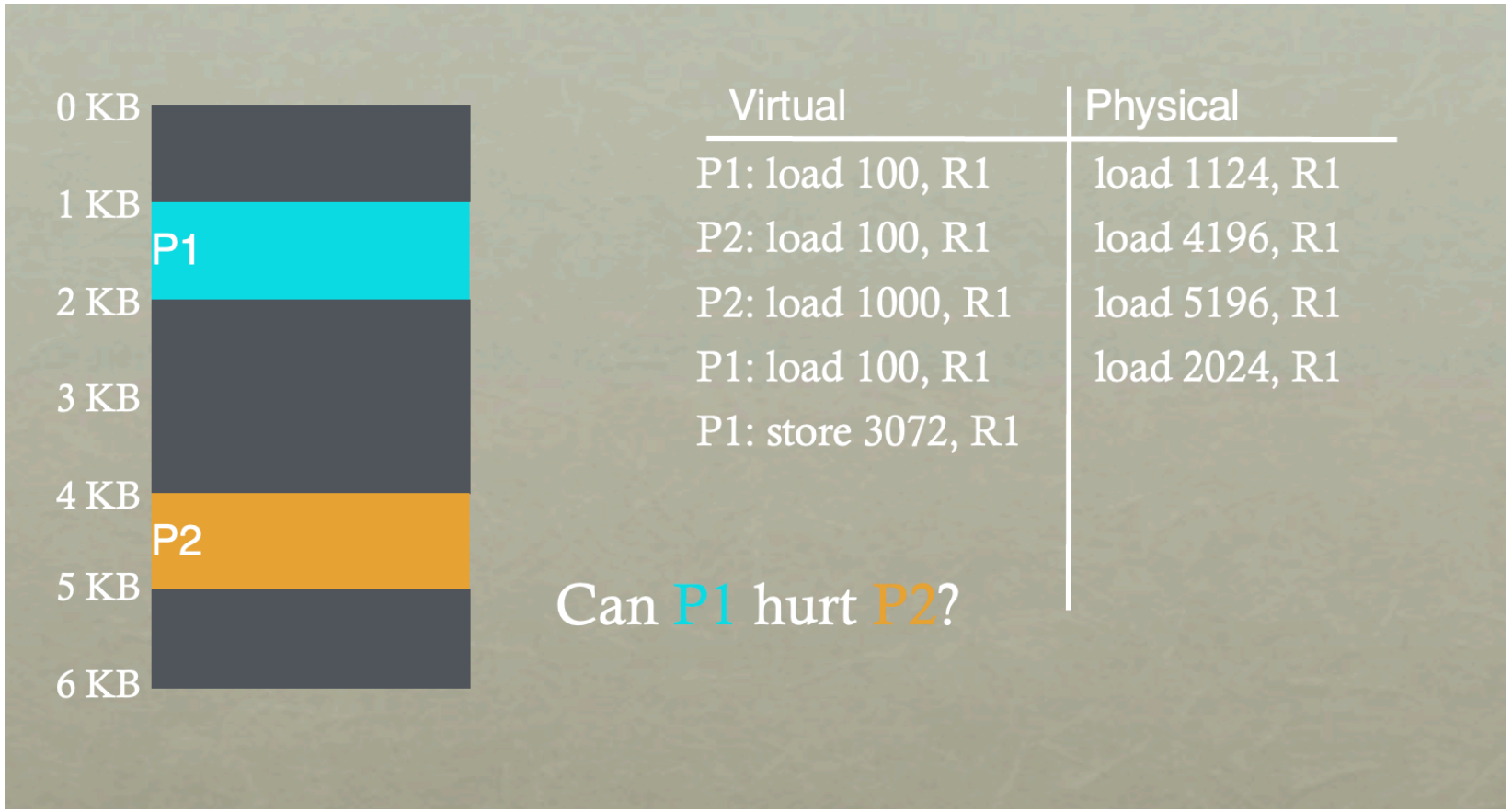
Translation on every memory access of user process

- 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




0 KB
1 KB
P1
2 KB
3 KB
4 KB
P2
5 KB
6 KB
base register
bounds register
P1 is running


0 KB
1 KB
P1
2 KB
3 KB
4 KB
P2
5 KB
6 KB
base register
bounds register
P2 is running



P2: load 1000, R1 => 4096+1000 = 5096 and not 5196

P1: load 1000, R1 => 1024+1000 = 2024. It should be load 1000, R1 and not load 100, R1.

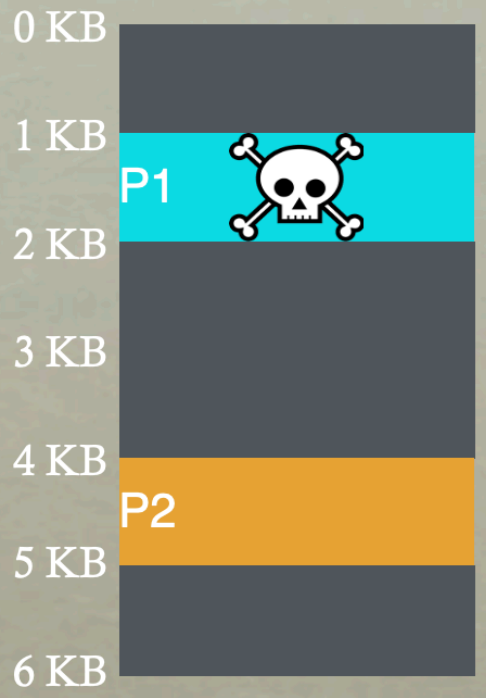
Virtual	Physical
P1: load 100, R1	load 1124, R1
P2: load 100, R1	load 4196, R1
P2: load 1000, R1	load 5196, R1
P1: load 100, R1	load 2024, R1
P1: store 3072, R1	interrupt OS!

3072 > 1024

Can P1 hurt P2?

P2: load 1000, R1
=> 4096+1000 =
5096 and not 5196

P1: load 1000, R1
=> 1024+1000 =
2024. It should be
load 1000, R1 and
not load 100, 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 100, R1	load 2024, R1
P1: store 3072, R1	interrupt OS!

Can P1 hurt P2?

P2: load 1000, R1 => 4096+1000 = 5096 and not 5196

P1: load 1000, R1 => 1024+1000 = 2024. It should be load 1000, R1 and not load 100, R1.

Managing Processes with Base and Bounds

Context-switch

- Store base and bounds registers in PCB (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

What if don't change base and bounds registers when switching?

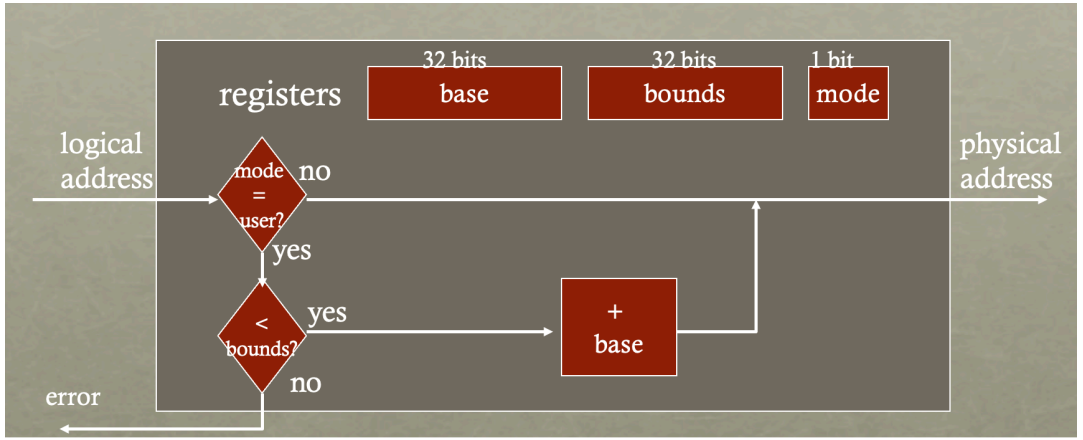
Protection requirement

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

Base and Bounds Advantages

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



Base and Bounds Disadvantages

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

Hardware Requirements	Notes
Privileged mode	*Needed to prevent user-mode processes from executing privileged operations*
Base/bounds registers	*Need pair of registers per CPU to support address translation and bounds checks*
Ability to translate virtual addresses and check if within bounds	*Circuitry to do translations and check limits; in this case, quite simple*
Privileged instruction(s) to update base/bounds	*OS must be able to set these values before letting a user program run*
Privileged instruction(s) to register exception handlers	*OS must be able to tell hardware what code to run if exception occurs*
Ability to raise exceptions	*When processes try to access privileged instructions or out-of-bounds memory*

Figure 15.3: **Dynamic Relocation: Hardware Requirements**

OS Requirements	Notes
Memory management	*Need to allocate memory for new processes;* *Reclaim memory from terminated processes;* *Generally manage memory via free list*
Base/bounds management	*Must set base/bounds properly upon context switch*
Exception handling	*Code to run when exceptions arise;* *likely action is to terminate offending process*

Figure 15.4: **Dynamic Relocation: Operating System Responsibilities**

OS @ boot (kernel mode)	Hardware	(No Program Yet)
initialize trap table		
	remember addresses of...	
	system call handler	
	timer handler	
	illegal mem-access handler	
	illegal instruction handler	
start interrupt timer		
	start timer; interrupt after X ms	
initialize process table		
initialize free list		

Figure 15.5: **Limited Direct Execution (Dynamic Relocation) @ Boot**

OS @ run (kernel mode)	Hardware	Program (user mode)
To start process A:		
allocate entry in process table		
alloc memory for process		
set base/bound registers		
return-from-trap (into A)		
	restore registers of A	
	move to **user mode**	
	jump to A's (initial) PC	
		Process A runs
		Fetch instruction
	translate virtual address	
	perform fetch	
		Execute instruction
	if explicit load/store:	
	ensure address is legal	
	translate virtual address	
	perform load/store	
		(A runs...)
	Timer interrupt	
	move to **kernel mode**	
	jump to handler	
Handle timer		
decide: stop A, run B		
call `switch()` routine		
save regs(A)		
to proc-struct(A)		
(including base/bounds)		
restore regs(B)		
from proc-struct(B)		
(including base/bounds)		
return-from-trap (into B)		
	restore registers of B	
	move to **user mode**	
	jump to B's PC	
		Process B runs
		Execute bad load
	Load is out-of-bounds;	
	move to **kernel mode**	
	jump to trap handler	
Handle the trap		
decide to kill process B		
deallocate B's memory		
free B's entry		
in process table		

Figure 15.6: **Limited Direct Execution (Dynamic Relocation) @ Runtime**

Segmentation

Divide address space into logical segments

- Each segment corresponds to logical entity in address space
- A segment is just a contiguous portion of the address space of a particular length
- Code, stack, heap

Each segment can independently:

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

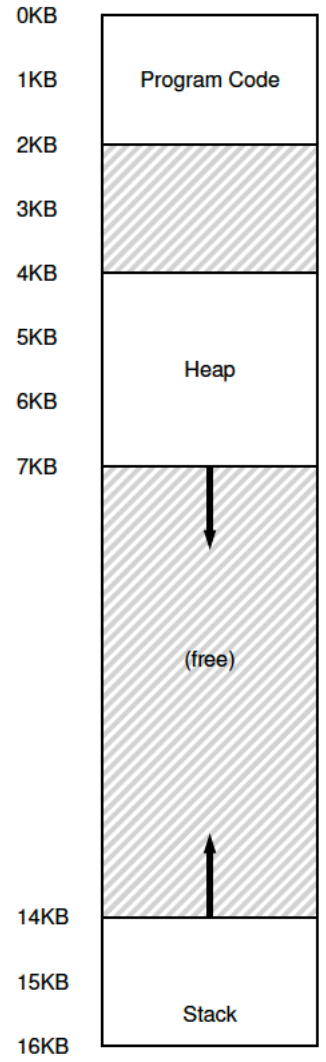
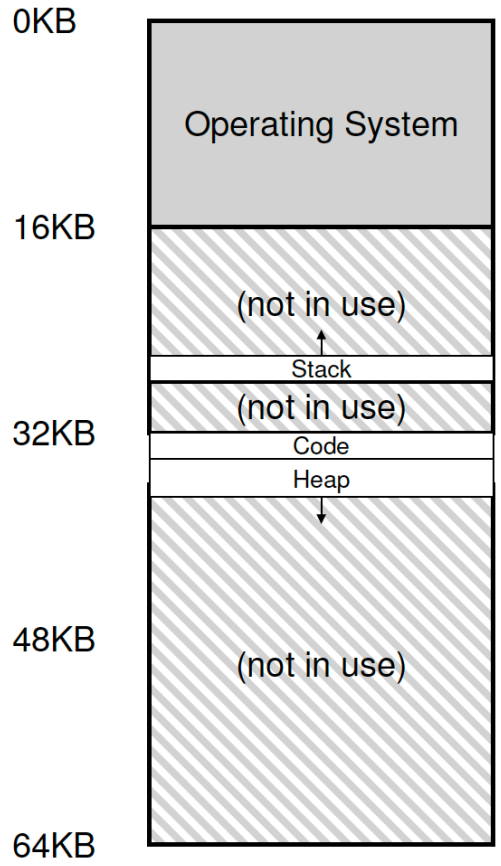


Figure 16.1: **An Address Space (Again)**

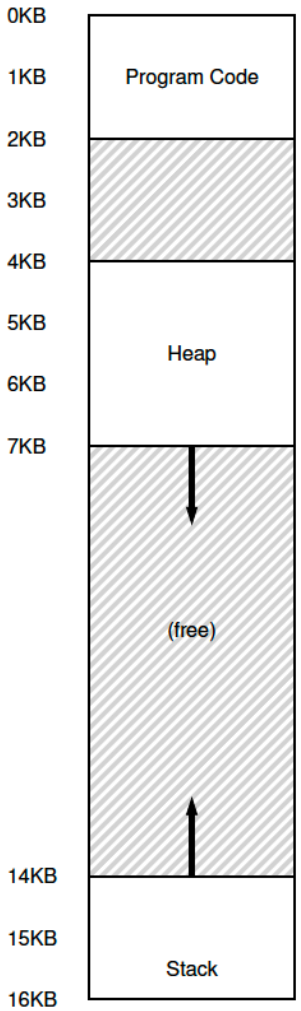
MMU Support for Segmentation

- MMU keeps separate base and bound register values for each segment of the process

Segment	Base	Size
Code	32K	2K
Heap	34K	3K
Stack	28K	2K



Example Translations for Segmentation



Segment	Base	Size
Code	32K	2K
Heap	34K	3K
Stack	28K	2K

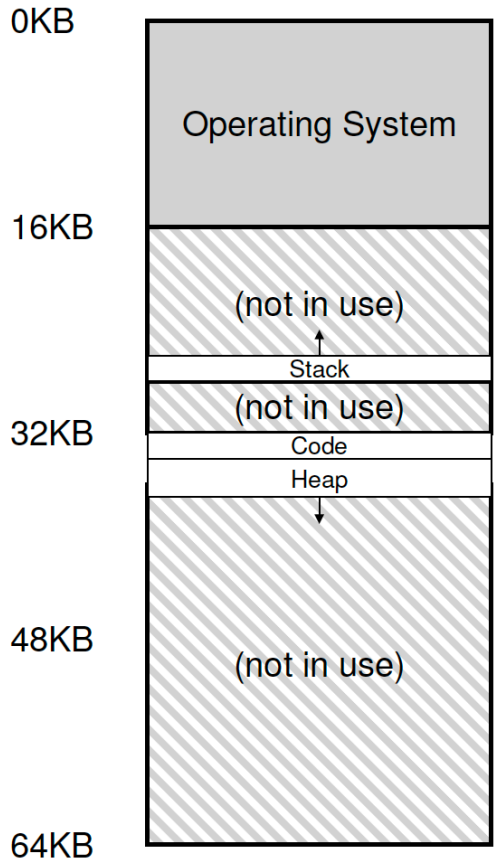
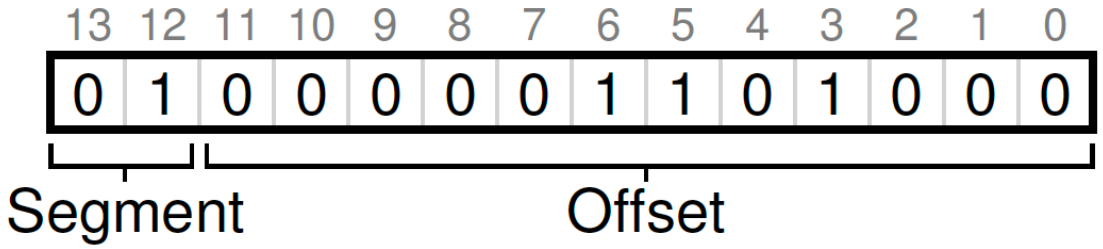


Figure 16.1: **An Address Space (Again)**

Which Segment Are We Referring To?

- 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 about the stack?

- MMU keeps track of whether the segment grows in the positive or negative direction

Segment	Base	Size (max 4K)	Grows Positive?
Code_{00}	32K	2K	1
Heap_{01}	34K	3K	1
Stack_{11}	28K	2K	0

Figure 16.4: **Segment Registers (With Negative-Growth Support)**

- Address translation for Stack is also slightly different.
- Example: 15 KB virtual address should map to ?

Support for Sharing

- Sometimes it is useful to share certain memory segments between address spaces.
  - Code sharing is common
- Hardware also stores **protection bits** per segment

Segment	Base	Size (max 4K)	Grows Positive?	Protection
Code_{00}	32K	2K	1	Read-Execute
Heap_{01}	34K	3K	1	Read-Write
Stack_{11}	28K	2K	0	Read-Write

Figure 16.5: **Segment Register Values (with Protection)**

OS Support for Segmentation

- What should the OS do on a context switch?
- What happens when a segment grows or shrinks?
- How to allocate new segments or grow existing ones?
  - External Fragmentation
  - Solution: Use a free-list management algorithm
    - Best-Fit
    - Worst-Fit
    - First-fit

Issue with Segmentation: Fragmentation

- Different-sized segments are allocated memory.
- Free memory (hole) is too small and scattered
  - Segment to be allocated may be larger than the free memory hole but smaller than the total amount of free memory available