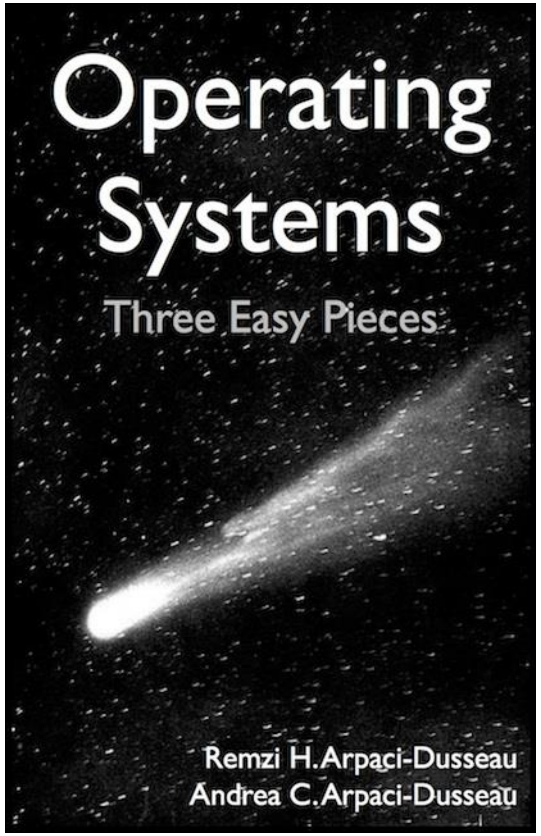
CMSC 125: Operating Systems

- Instructor: **Joseph Anthony C. Hermocilla**
- Email: jchermocilla@up.edu.ph
- Web: https://jachermocilla.org

Resources

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

Slides Template:
https://pages.cs.wisc.edu/~remzi/OSTEP/Educators-Slides/Youjip/



Acknowledgement

- This lecture slide set was initially developed for Operating System course in Computer Science Dept. at Hanyang University. This lecture slide set is for OSTEP book written by Remzi and Andrea at University of Wisconsin.

15. Address Translation

Operating System: Three Easy Pieces

Memory Virtualizing with Efficiency and Control + Flexibility

- Memory virtualizing takes a similar strategy known as **limited direct execution(LDE)** for **efficiency** and **control**.
- In memory virtualizing, efficiency and control are attained by hardware support.
    - e.g., registers, TLB(Translation Lookaside Buffer)s, page table
- Also provide **flexibility** – allow processes to be able to use their address space in whatever way they like

Address Translation

- Hardware transforms a **virtual address** to a **physical address**.
    - The desired information is actually stored in a physical address.

- The OS must get involved at key points to set up the hardware
    - The OS must **manage memory,** to judiciously intervene
- Current Assumptions:
    1. User's address space is placed **contiguously** in physical memory
    2. Address space size is **not too big**, less than the size of the physical memory
    3. Address space size for all processes are **the same**

Example: Address Translation

- C - Language code

```
void func()
    int x = 3000;
    x = x + 3; // this is the line of code we are interested in
    ...
```

- **Load** a value from memory
- **Increment** it by three
- **Store** the value back into memory

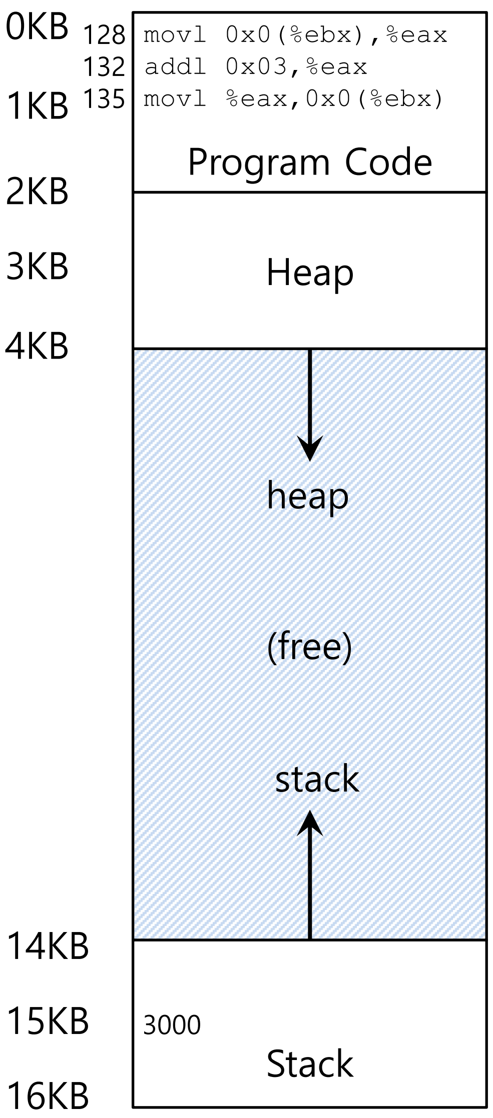
Example: Address Translation(Cont.)

- Assembly

```
128 : movl 0x0(%ebx), %eax          ; load 0+ebx into eax
132 : addl $0x03, %eax              ; add 3 to eax register
135 : movl %eax, 0x0(%ebx)          ; store eax back to mem
```

  - Presume that the address of `‘x’` has been place in `ebx` register.
  - **Load** the value at that address into `eax` register.
  - **Add** 3 to `eax` register.
  - **Store** the value in `eax` back into memory.

Example: Address Translation(Cont.)

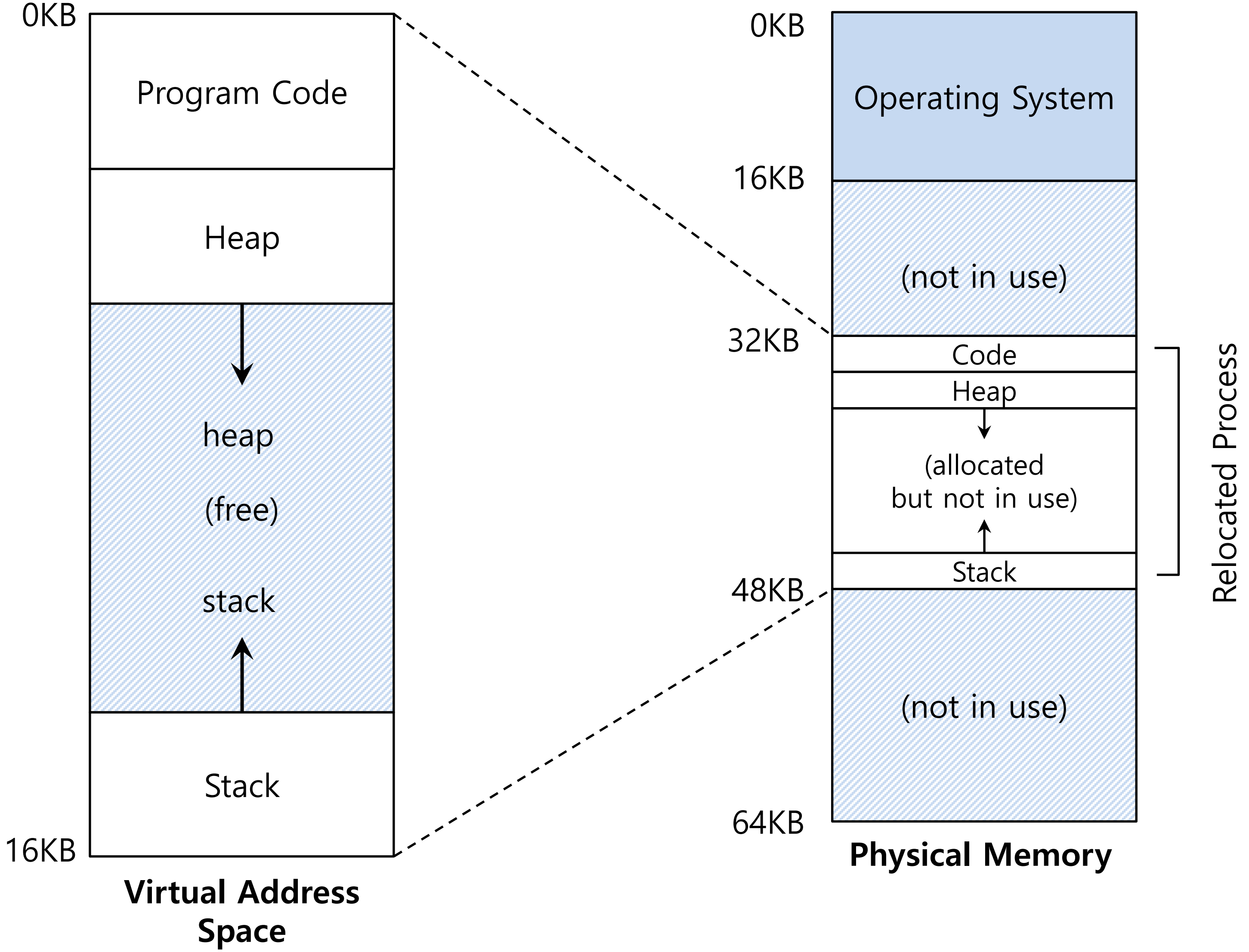


- Fetch instruction at address 128
- Execute this instruction (load from address 15KB)
- Fetch instruction at address 132
- Execute this instruction (no memory reference)
- Fetch the instruction at address 135
- Execute this instruction (store to address 15 KB)

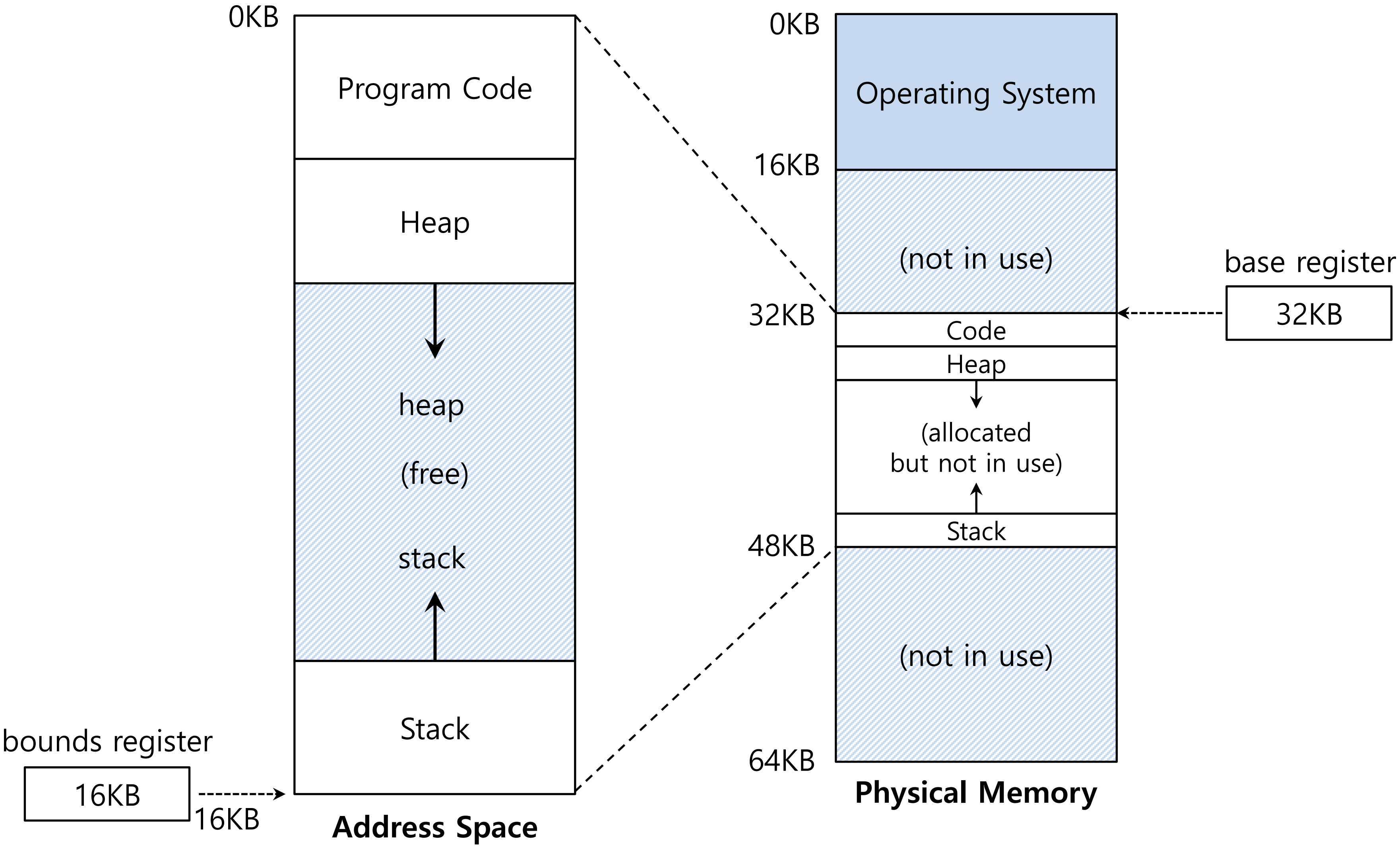
Address Space Relocation

- The process' view of its **virtual address space** is from 0 to 16KB
  - It *might* be the case that this maps exactly to physical memory
- Say, the OS wants to place the process **somewhere else** in physical memory, not at address 0?
  - Can this be done without changing the process's view of its virtual address space?

A Single Relocated Process



Dynamic Relocation(Hardware-based): Using Base and Bounds Registers



Dynamic Relocation(Hardware-based): Using Base and Bounds Registers (Cont..)

- Program is written and compiled as if it is loaded at address zero(0)
- When a program starts running, the OS decides **where** in physical memory a process should be **loaded**
  - Set the **base register** a value.

$$\boxed{phycal\ address = virtual\ address + base}$$

  - Every virtual address must **not be greater than bound** and **not negative.**

$$\boxed{0 \leq\ virtual\ addressvirtual\ address < bounds}$$

Dynamic Relocation(Hardware-based): Using Base and Bounds Registers (Cont..)

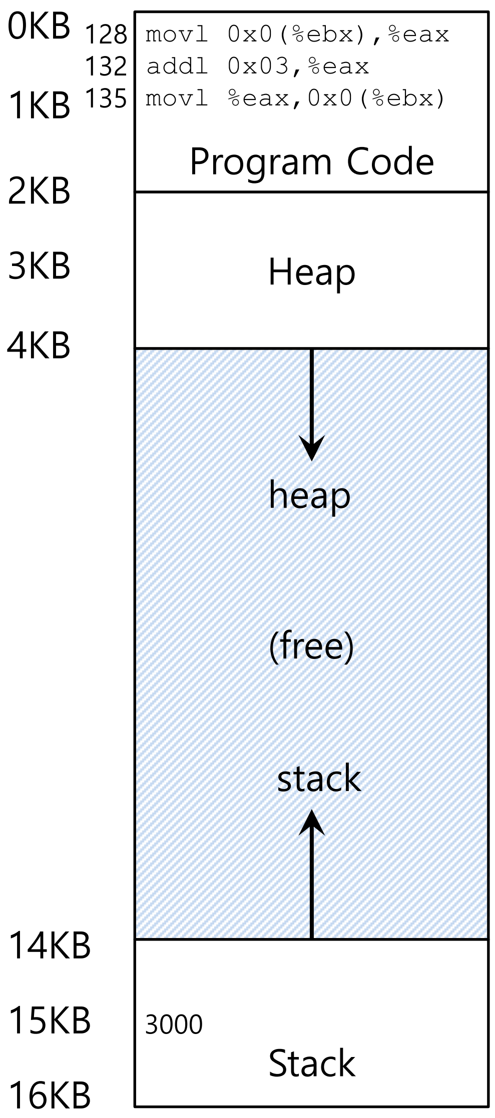
```
128 : movl 0x0(%ebx), %eax
```

- **Fetch** instruction at address 128

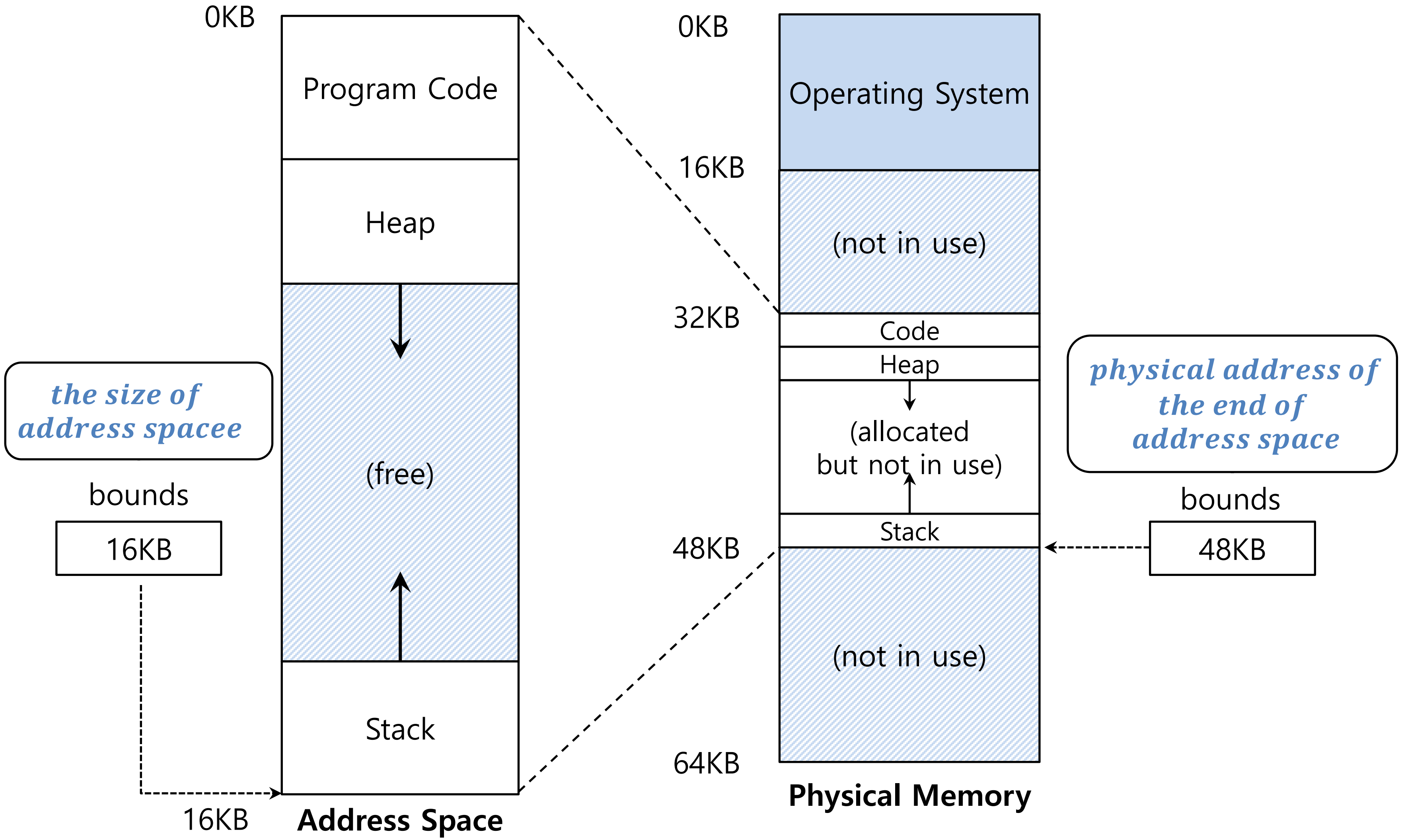
$$32896 = 128 + 32KB(base)$$

- **Execute** this instruction
  - Load from address 15KB

$$47KB = 15KB + 32KB(base)$$

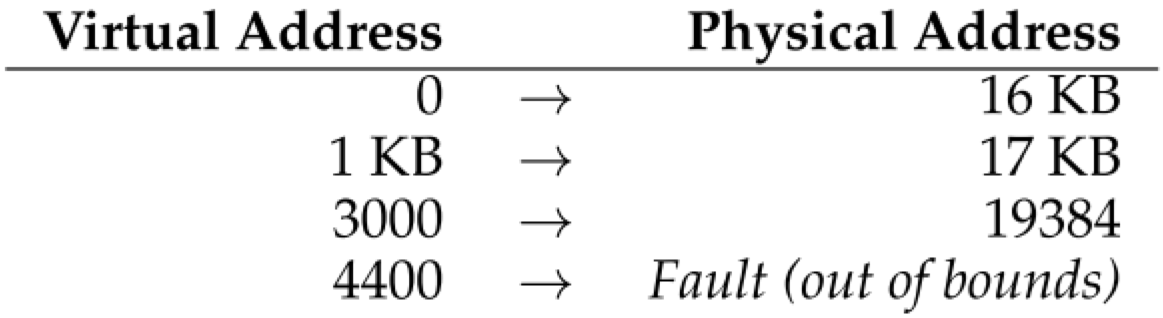


Two ways to define the Bounds Register



Example Translations

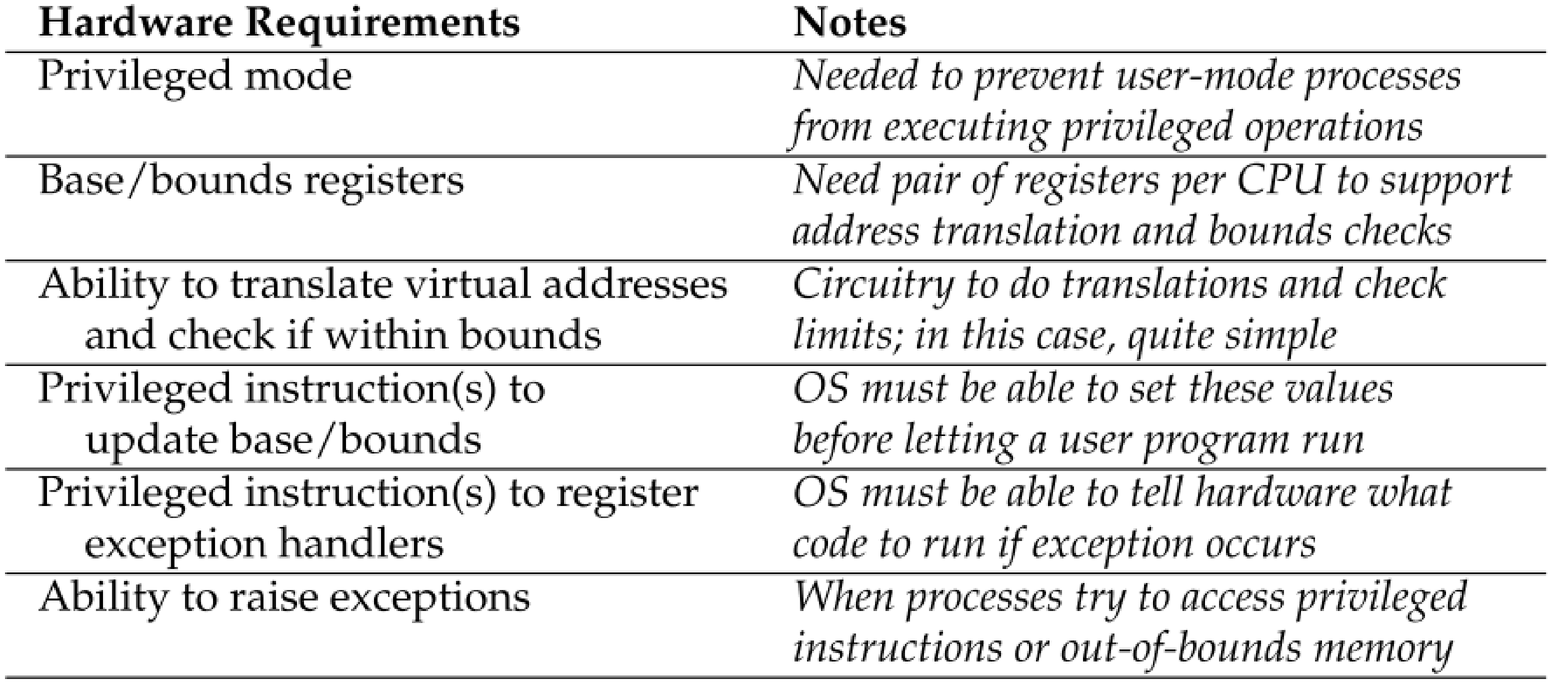
- Given a process with:
  - Address Space Size = 4KB
  - Loaded at physical address 16KB
- Sample address translations



Virtual Address		Physical Address
0	→	16 KB
1 KB	→	17 KB
3000	→	19384
4400	→	*Fault (out of bounds)*

Summary of hardware support needed for Dynamic Relocation

- Processor modes: Kernel Mode and User Mode determined through a **processor status word**
- Memory Management Unit: **Base Register** and **Bounds Register**
- Changing the base and bounds registers should be allowed only in Kernel Mode
  - When scheduler switches processes
- Processor should be able to generate **exceptions** during illegal memory access



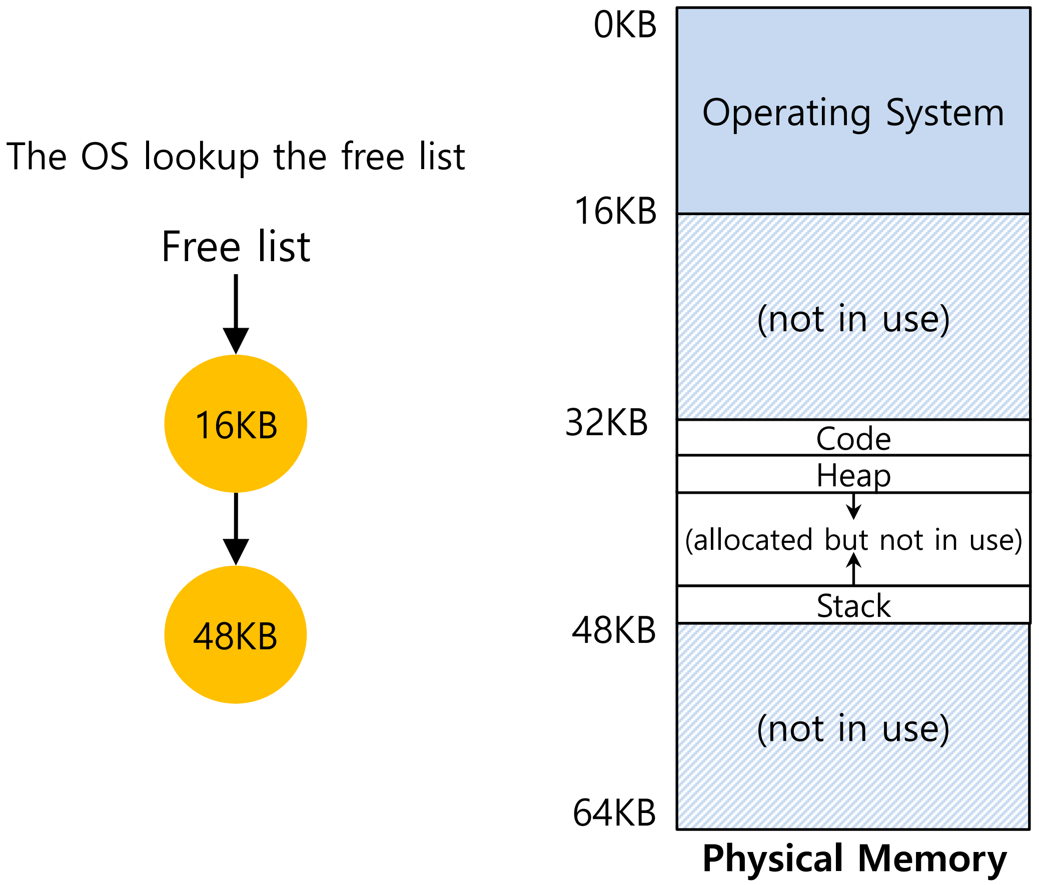
Hardware Requirements	**Notes**
Privileged mode	*Needed to prevent user-mode processes from executing privileged operations*
Base/bounds registers	*Need pair of registers per CPU to support address translation and bounds checks*
Ability to translate virtual addresses and check if within bounds	*Circuitry 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*

Issues that the OS must address for Dynamic Relocation

- The OS must **take action** to implement **base-and-bounds** approach
- Three critical junctures:
  - When a process **starts running**
    - Finding space for address space in physical memory, maintain a **free list**
  - When a process is **terminated**
    - Reclaiming the memory for use by other processes
  - When context **switch occurs**
    - Saving and storing the base-and-bounds register pair for each process since we only have one pair per core
    - When a process is blocked, it can easily be moved to a different location
  - At boot time, the OS must set exception handlers using privileged instructions

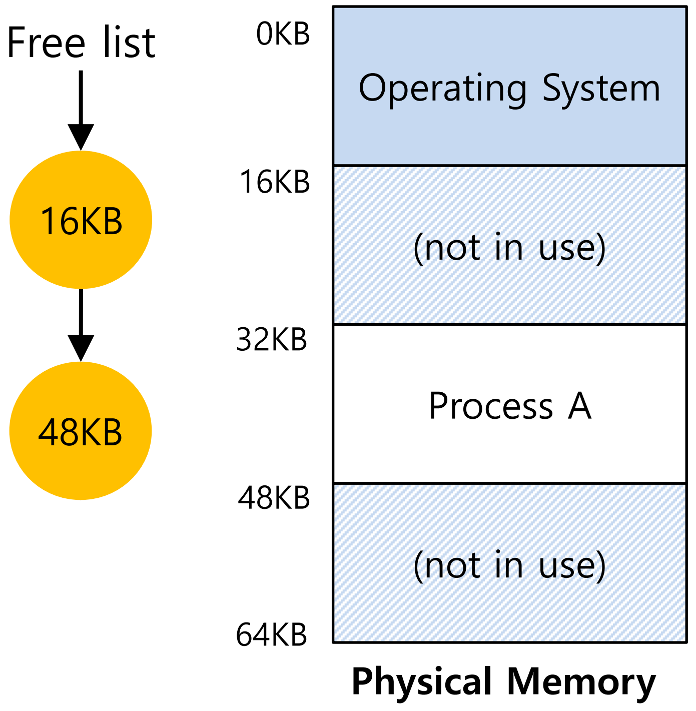
OS Issues: When a Process Starts Running

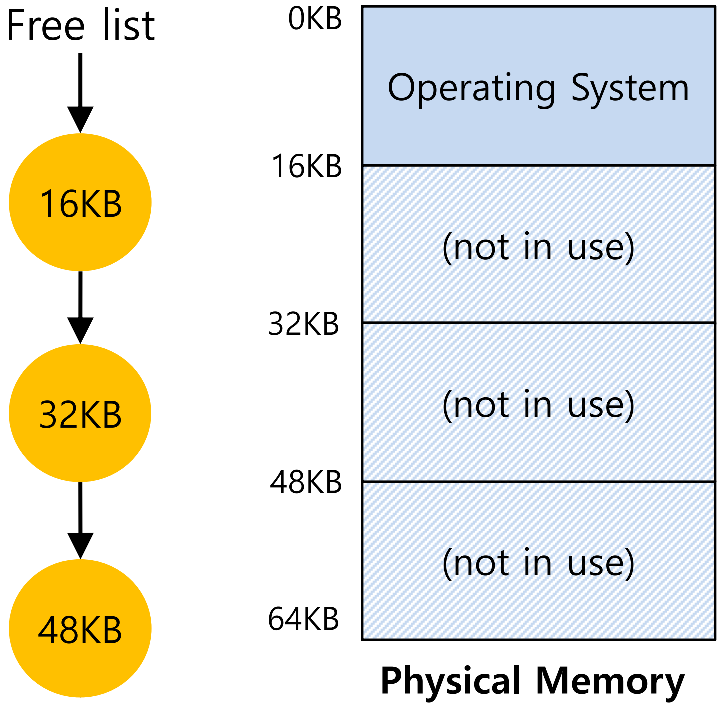
- The OS must **find a room** for a new address space
  - **free list** : A list of the range of the physical memory which are not in use



OS Issues: When a Process Is Terminated

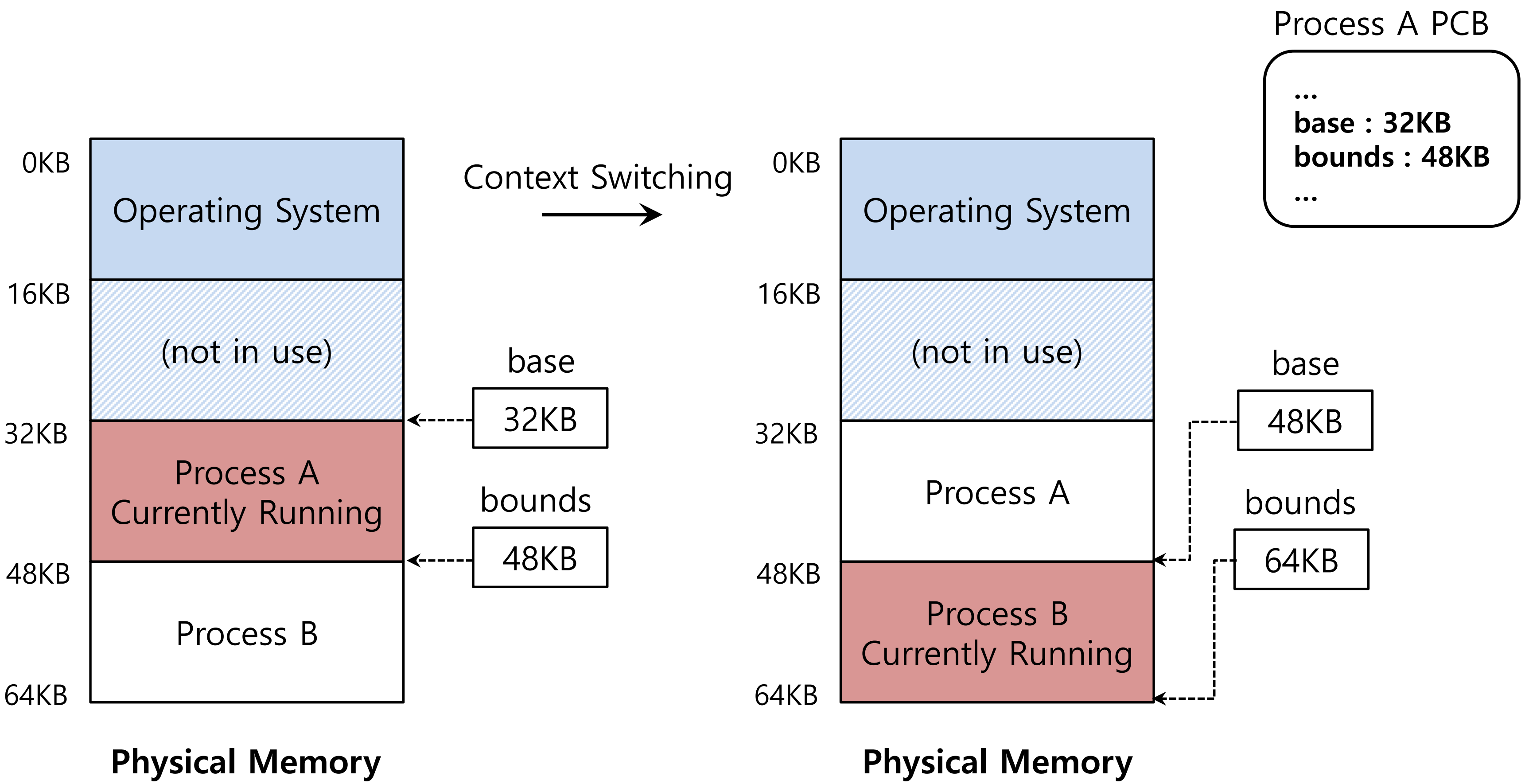
- The OS must **put the memory back** on the free list





OS Issues: When Context Switch Occurs

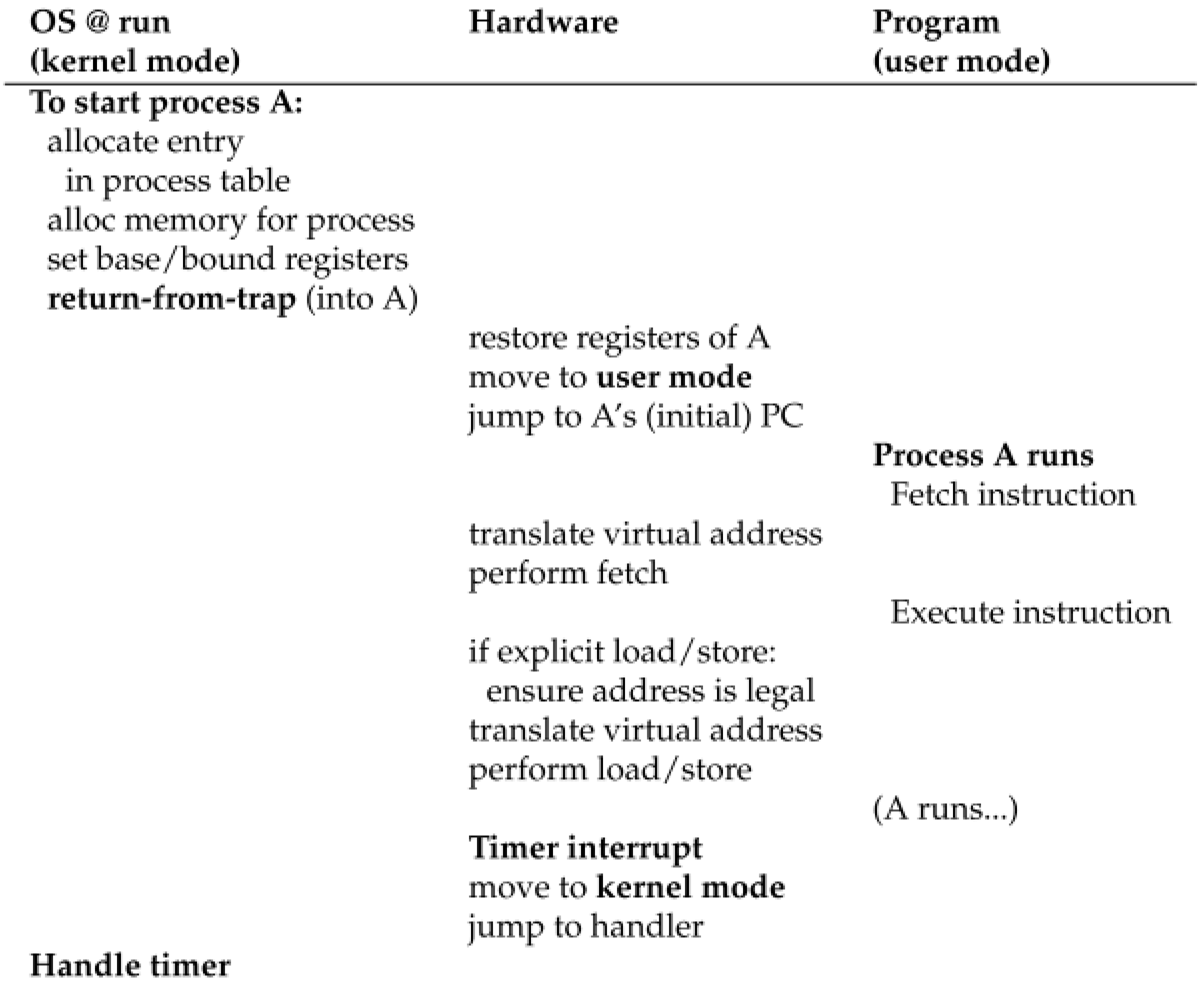
- The OS must **save and restore** the base-and-bounds pair.
  - In **process structure** or **process control block(**PCB)



OS Issues: Limited Direct Execution with Dynamic Relocation at Boot Time

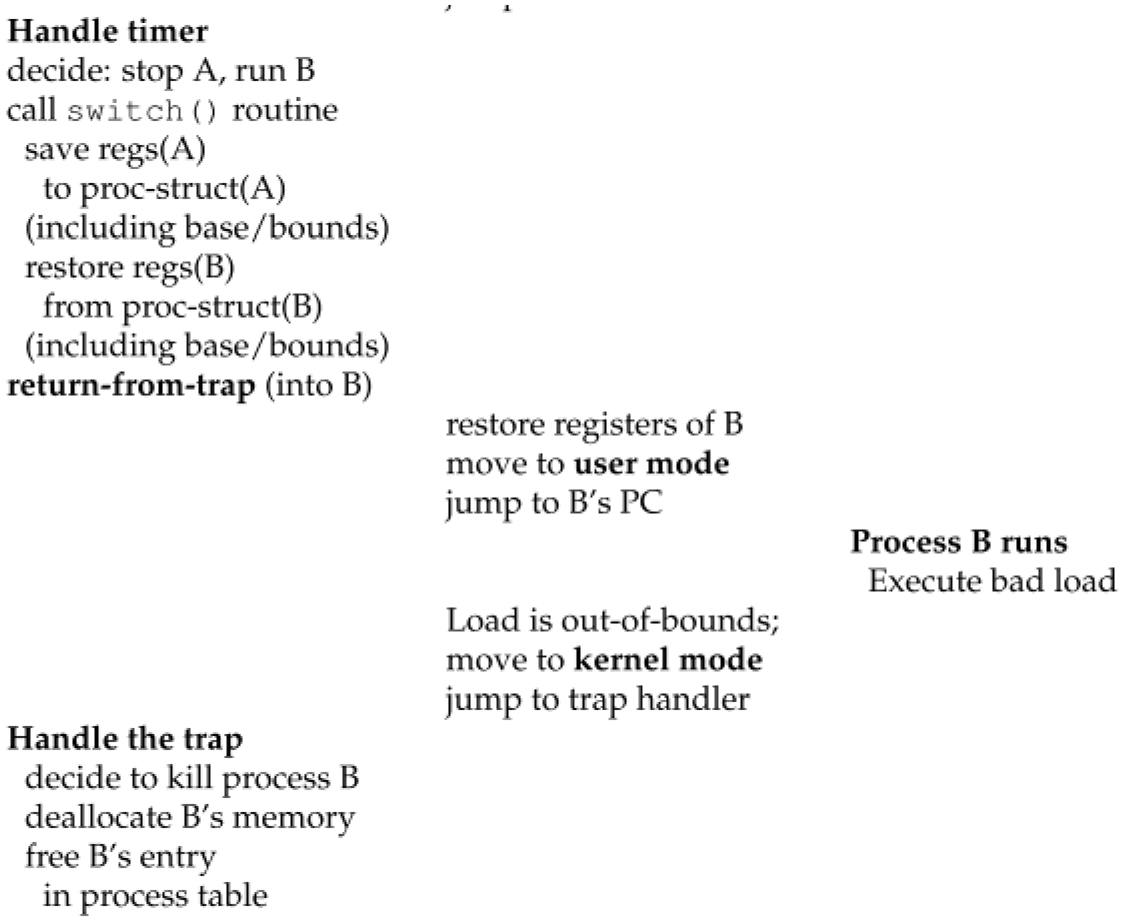
OS @ boot **(kernel mode)**	**Hardware**	**(No Program Yet)**
initialize trap table		
	remember addresses of... system call handler timer handler illegal mem-access handler illegal instruction handler	
start interrupt timer		
	start timer; interrupt after X ms	
initialize process table		
initialize free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

OS Issues: Limited Direct Execution with Dynamic Relocation at Runtime



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		

OS Issues: Limited Direct Execution with Dynamic Relocation at Runtime (Cont..)



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		

Other Issues with Dynamic Relocation

- **Internal Fragmentation** – not all allocated space is used