# 15. Address Translation

## Operating System: Three Easy Pieces

# Memory Virtualizing with Efficiency and Control

- 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 Look-aside Buffer)s, page-table

# Assumptions Made (just in this lecture)

- User address space is contiguous in memory
- User address space is smaller than the physical memory (up to 64KB)
- Each address space has the same size (up to 16KB)

# 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.

# Example: Address Translation

- C - Language code

```
void func()
        int x;
        ...
        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

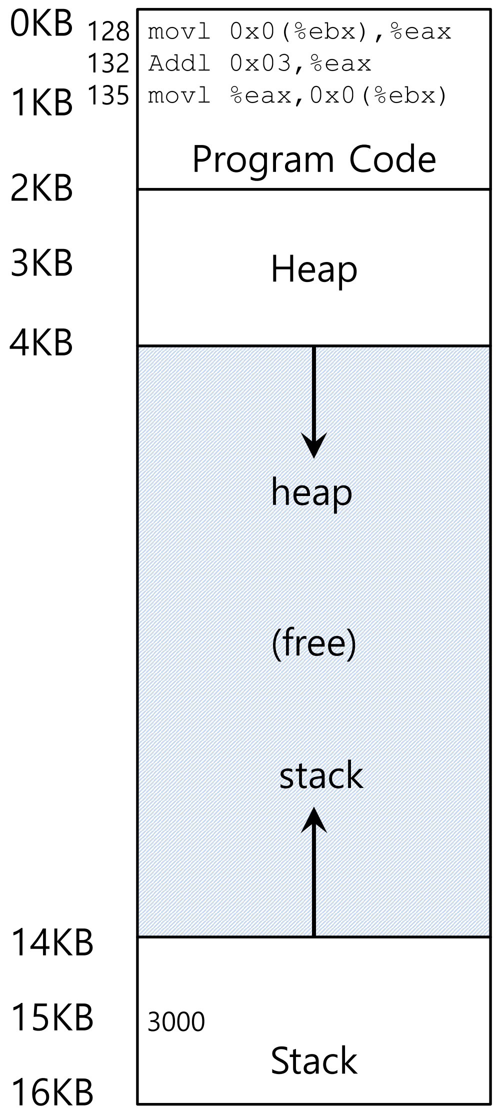
# Example: Address Translation(Cont.)

- 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
```

  - **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.)



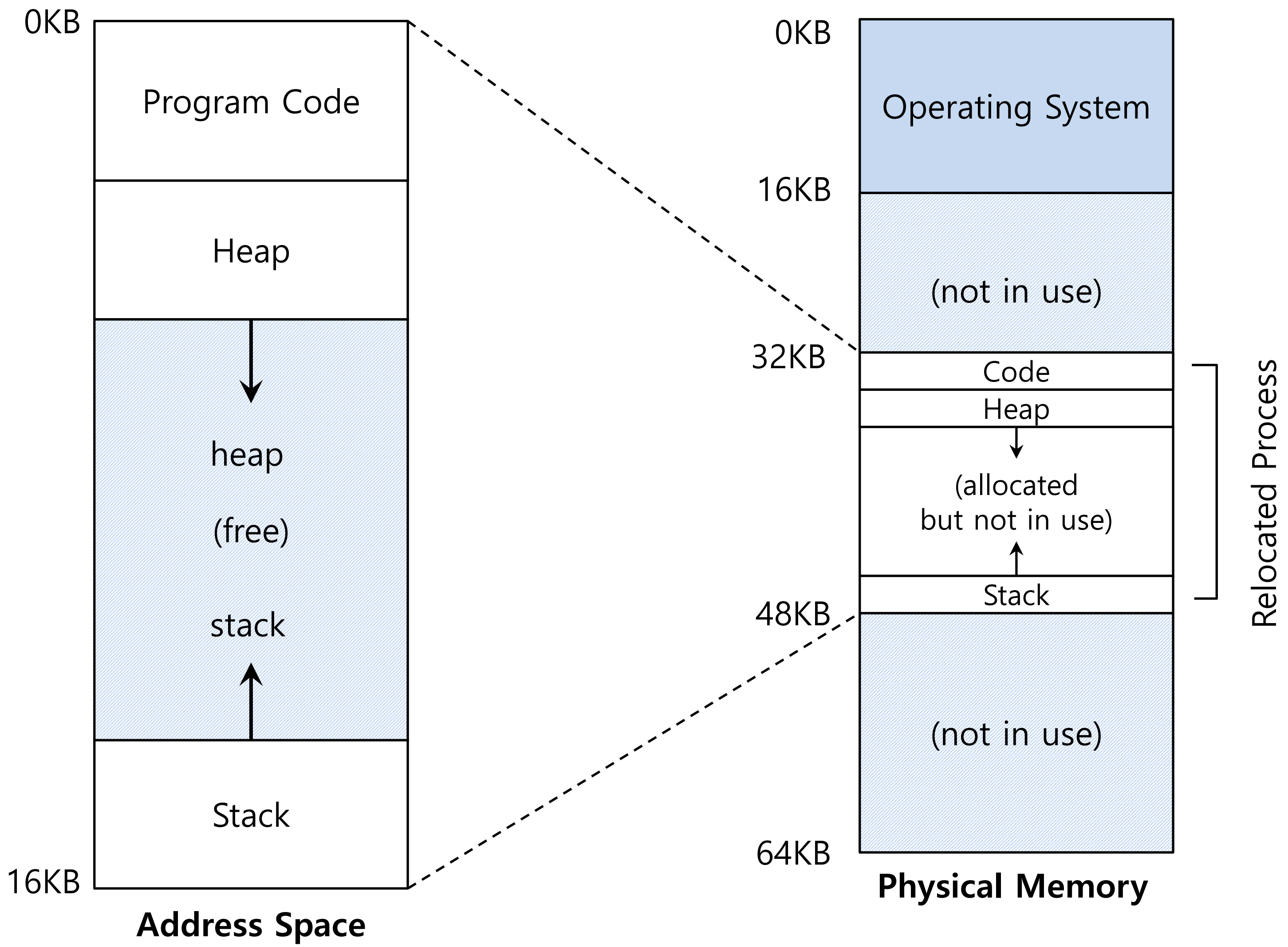

- Fetch instruction at address 128
- Execute this instruction (load from address 15KB)
- Fetch instruction at address 132
- Execute this instruction (no memory reference)
- Fetch the instruction at address 135
- Execute this instruction (store to address 15 KB)

# Relocation Address Space

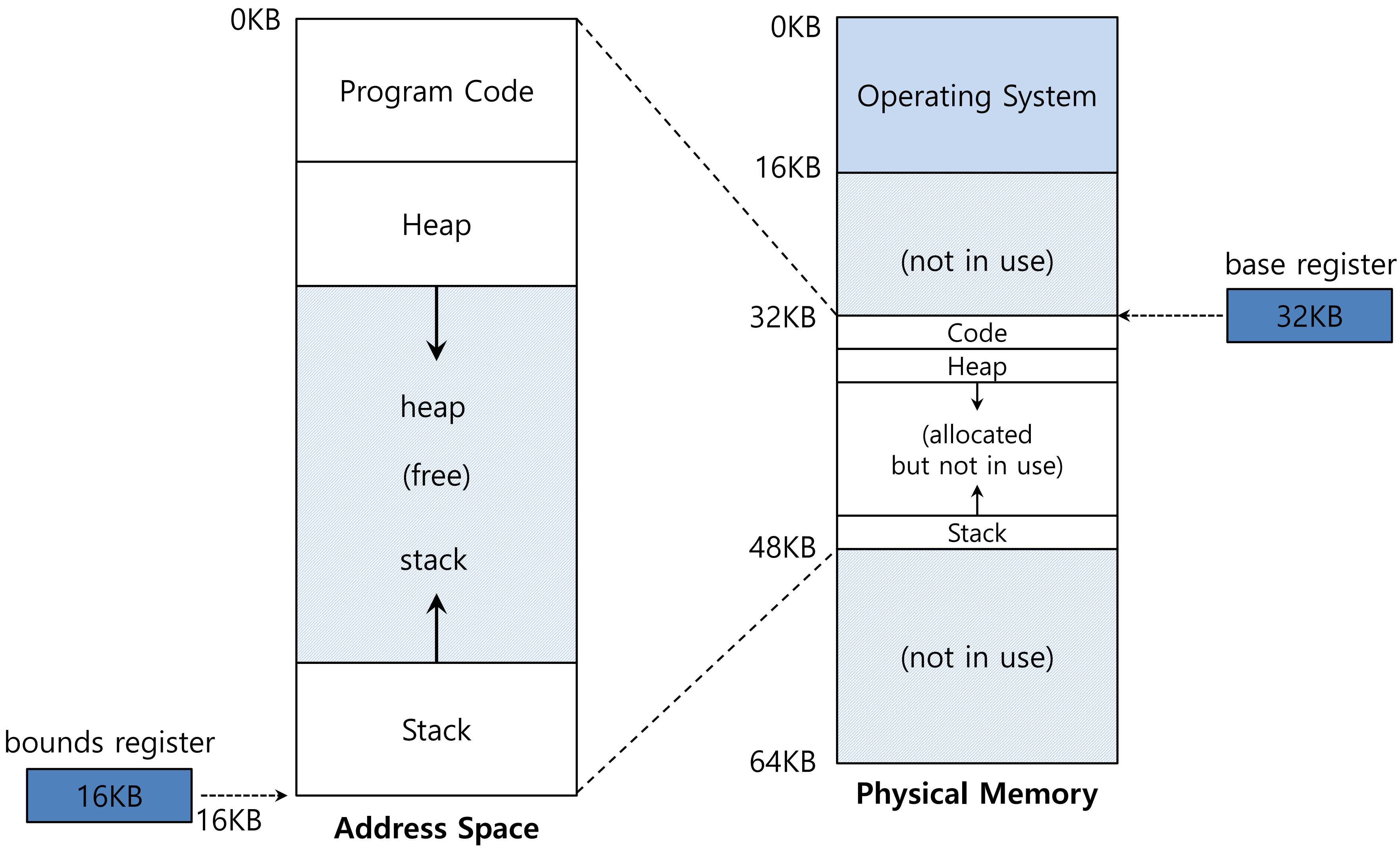
- The OS wants to place the process **somewhere else** in physical memory, not at address 0.
  - The address space start at address 0.

- But how make it **transparently**?

# A Single Relocated Process



# Base and Bounds Registers



# Aside: Software-based Relocation

- If Harware-support is not present?
  - Static-Relocation
- Loader should “transform” the executable
- Problems
  - No protection
  - Late relocation is hard
- **Hardware support is mandatory!**

# Dynamic (Hardware based) Relocation

- When a program starts running, the OS decides **where** in physical memory a process should be **loaded**.
  - Set the **base** register a value.

$$phycal\ address = virtual\ address + base$$

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

$$0 \leq\ virtual\ address < bounds$$

  - ISA provide
    - **Privileged** ins to handle those registers
    - Specific **exception** to detect (and handle miss behaviors)

# Relocation and Address Translation

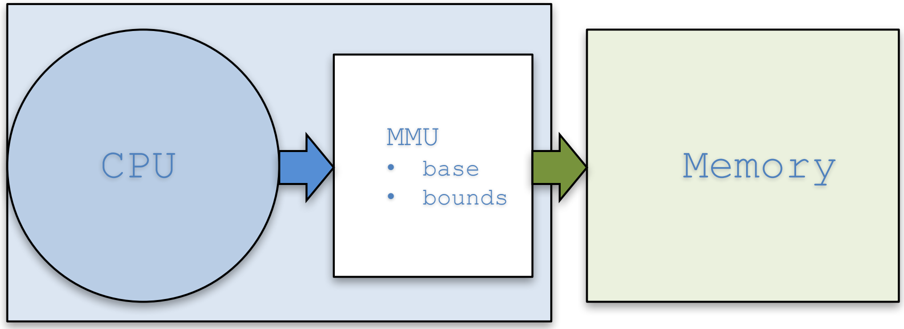
```
128 : movl 0x0(%ebx), %eax
```

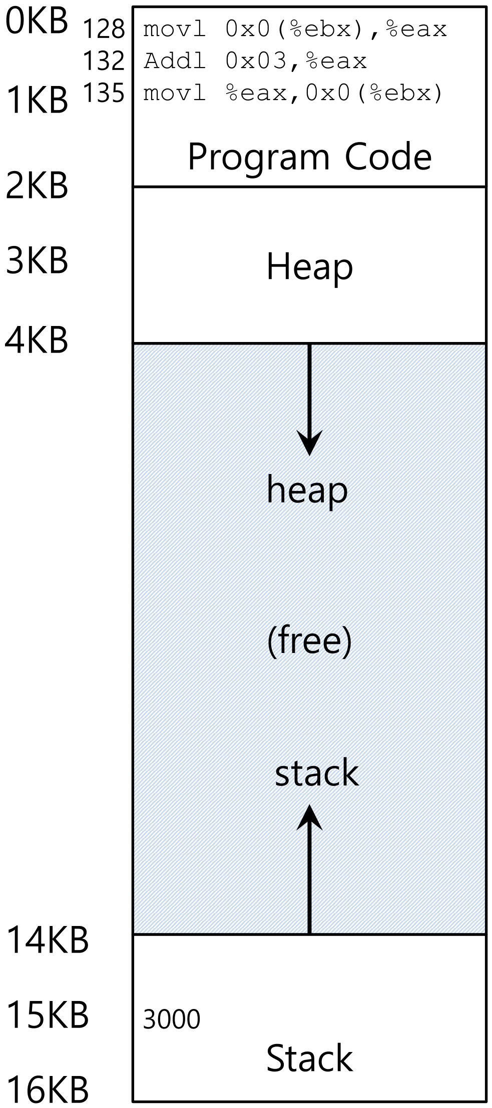
- **Fetch** instruction at address 128

$32896 = 128 + 32KB(base)$

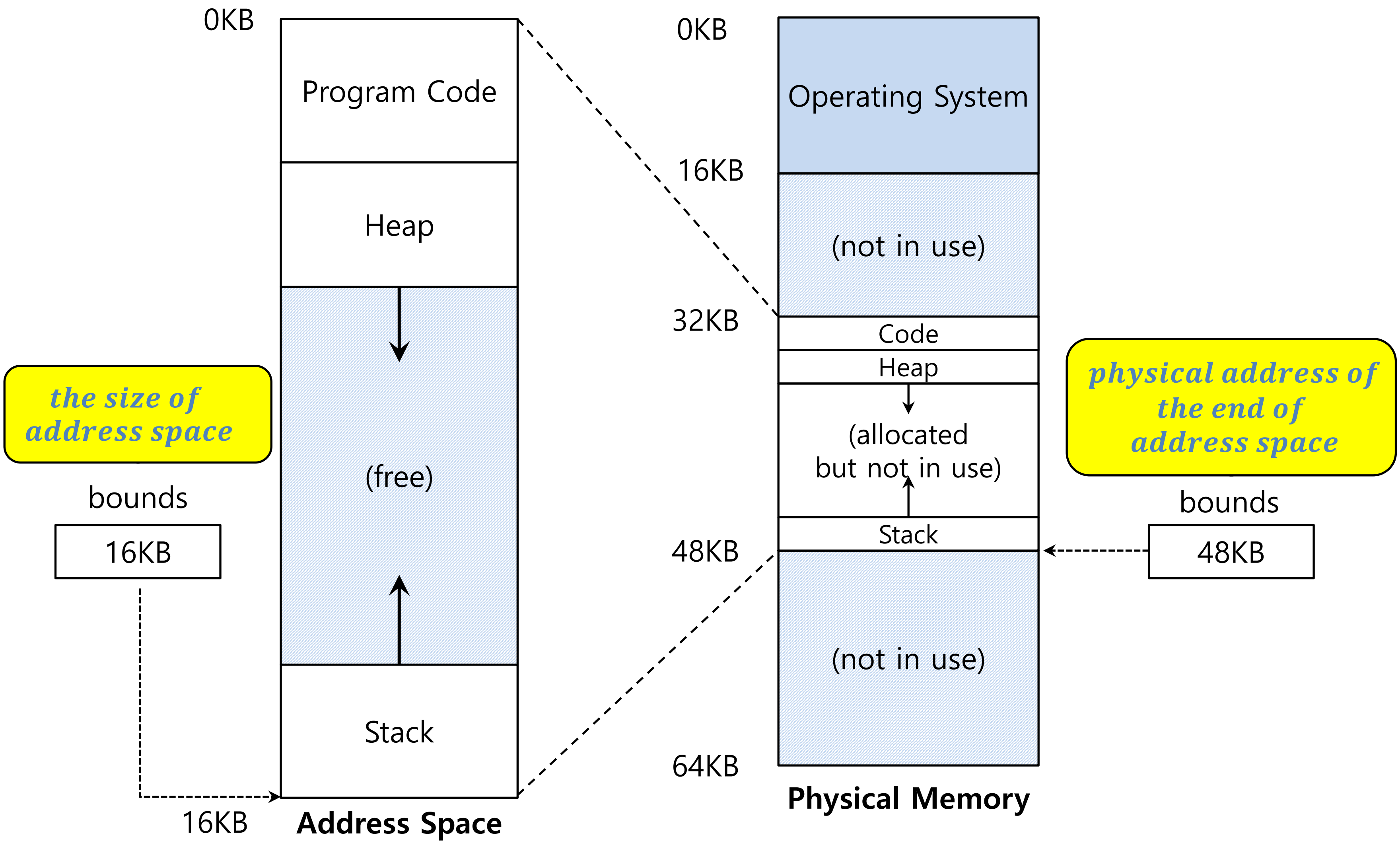
- **Execute** this instruction
  - Load from address 15KB

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



# Two ways of Bounds Register

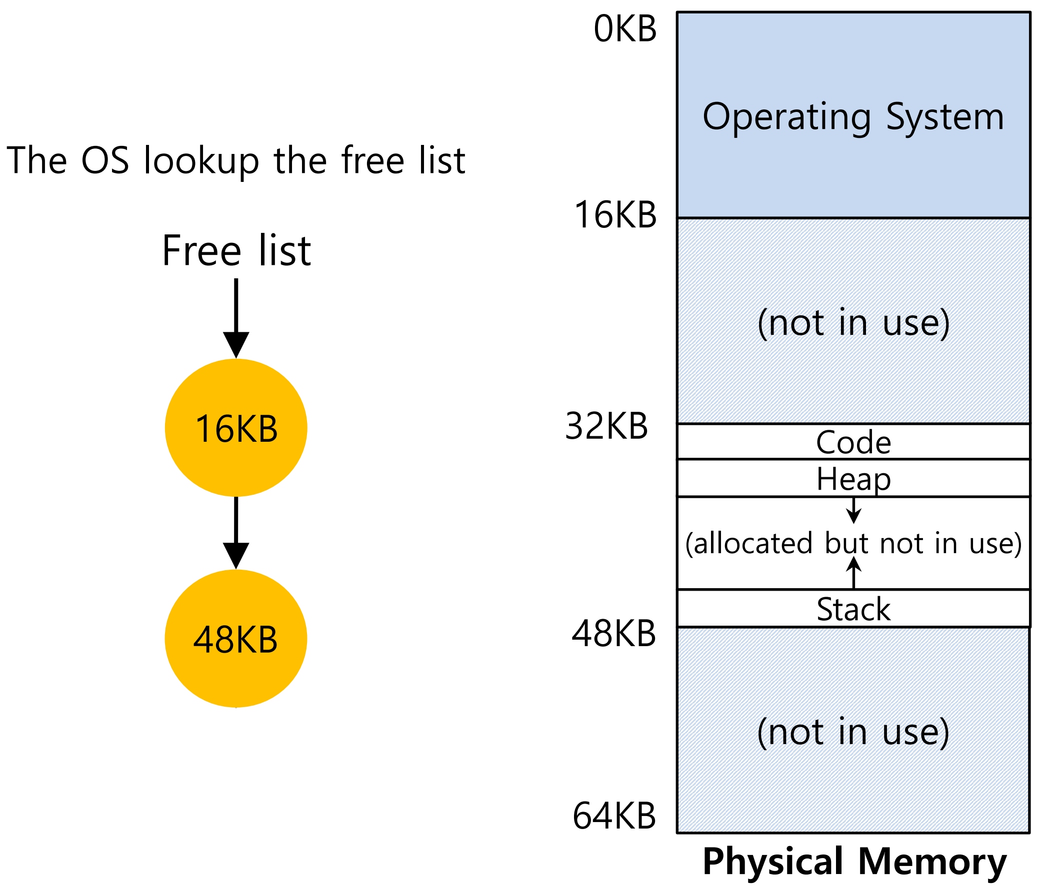


# OS Issues for Memory Virtualizing

- 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
  - When a process is **terminated**:
    - Reclaiming the memory for use
  - When context **switch occurs**:
    - Saving and storing the base-and-bounds pair

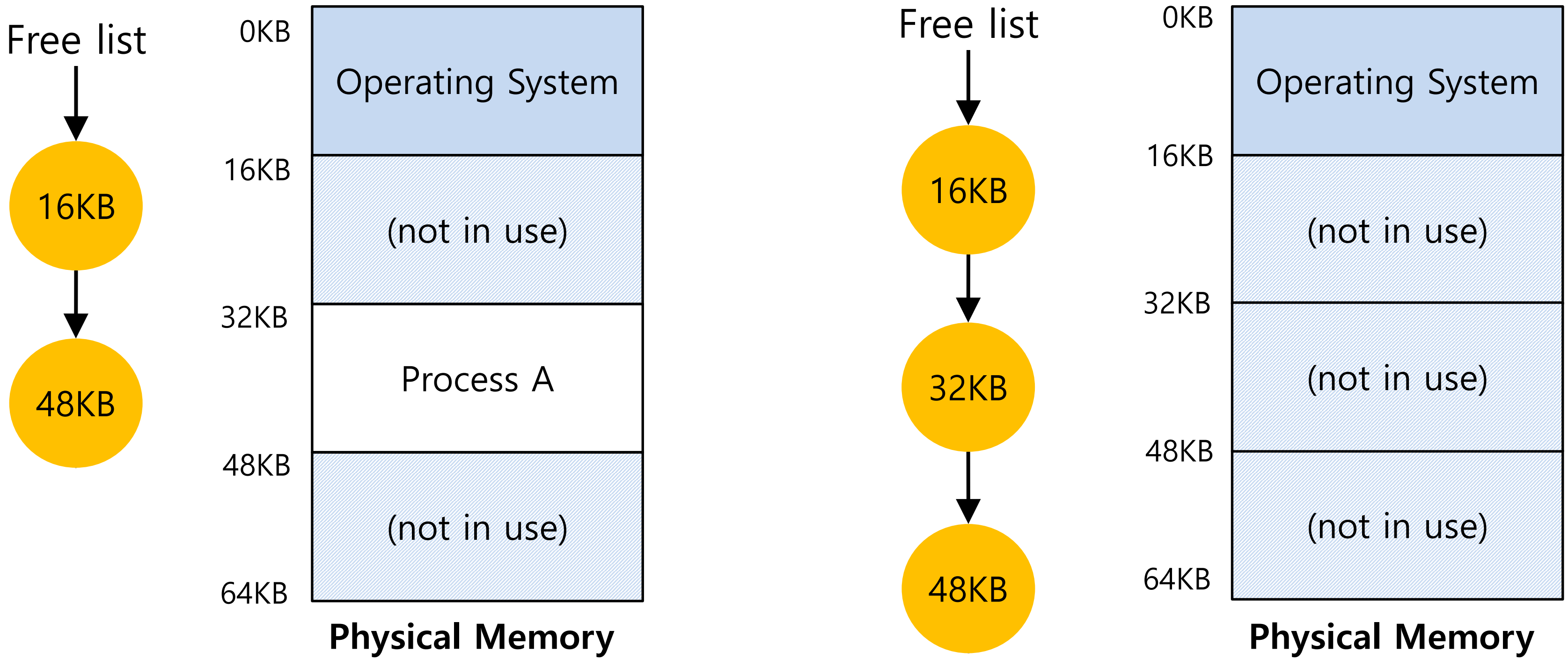
# OS Issues: When a Process Starts Running

- 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.



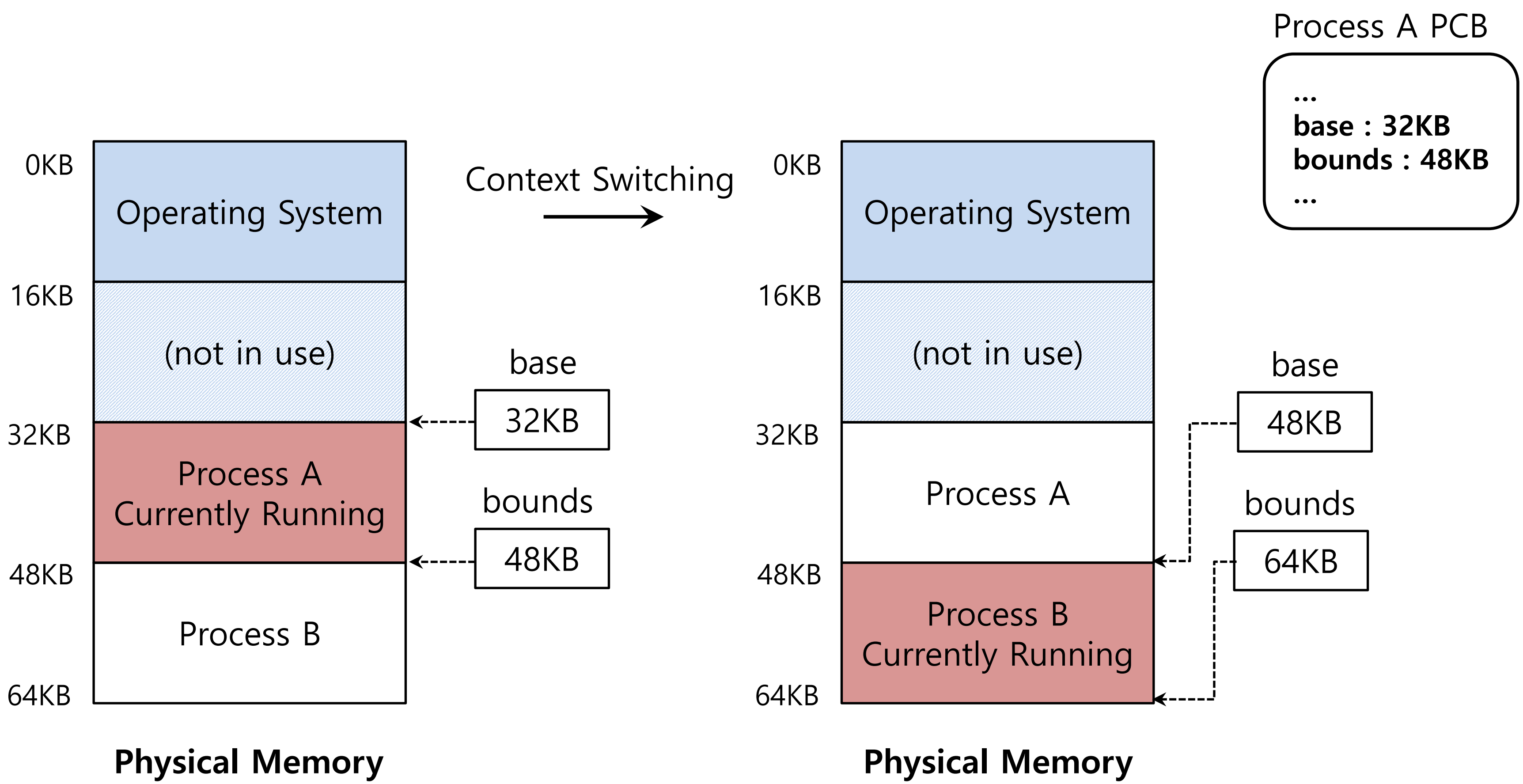
# OS Issues: When a Process Is Terminated

- 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)



# Summary: Dynamic Relocation (strawman version)

| **OS @ boot (kernel mode)** | **Hardware** |
|---|---|
| **initialize trap table** | |
| | remember addresses of...<br>system call handler<br>timer handler<br>illegal mem-access handler<br>illegal instruction handler |
| **start interrupt timer** | |
| | start timer; interrupt after X ms |
| **initialize process table**<br>**initialize free list** | |

| **OS @ run (kernel mode)** | **Hardware** | **Program (user mode)** |
|---|---|---|
| **To start process A:**<br>allocate entry in process table<br>allocate memory for process<br>set base/bounds registers<br>**return-from-trap** (into A) | | |
| | restore registers of A<br>move to **user mode**<br>jump to A's (initial) PC | |
| | | **Process A runs**<br>Fetch instruction |
| | Translate virtual address<br>and perform fetch | |
| | | Execute instruction |

# Summary: Dynamic Relocation (strawman version)

Fetch instruction

Translate virtual address
and perform fetch

Execute instruction

If explicit load/store:
Ensure address is in-bounds;
Translate virtual address
and perform load/store

...

**Timer interrupt**
move to **kernel mode**
Jump to interrupt handler

**Handle the trap**
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 terminate process B
de-allocate B's memory
free B's entry in process table

- Disclaimer: Disclaimer: This lecture slide set is used in AOS course at University of Cantabria. 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 Arpaci-Dusseau (at University of Wisconsin)