Operating Systems

Lecture 9

13. The Abstraction: Address Space

Memory Virtualization

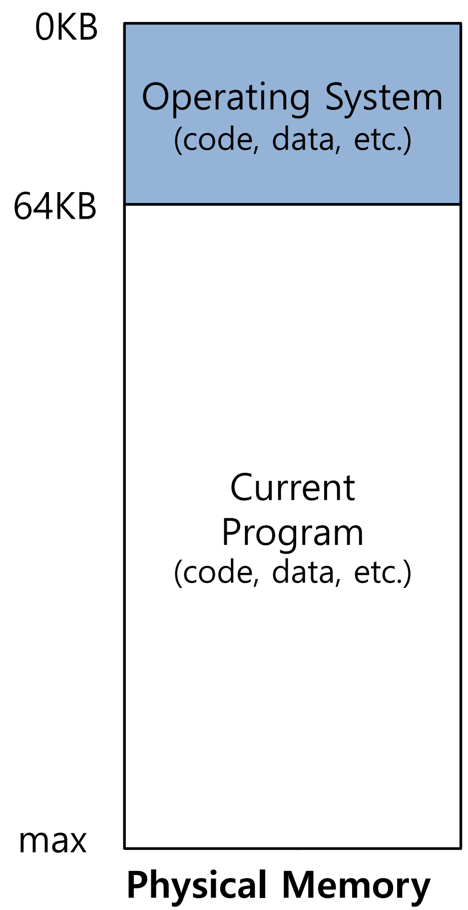
- What is **memory virtualization**?
  - OS virtualizes its physical memory.
  - OS provides an illusion memory space per each process.
  - It seems to be seen like each process uses the whole memory.

Benefit of Memory Virtualization

- Ease of use in programming
- Memory efficiency in terms of times and space
- The guarantee of isolation for processes as well as OS
  - Protection from errant accesses of other processes

OS in The Early System

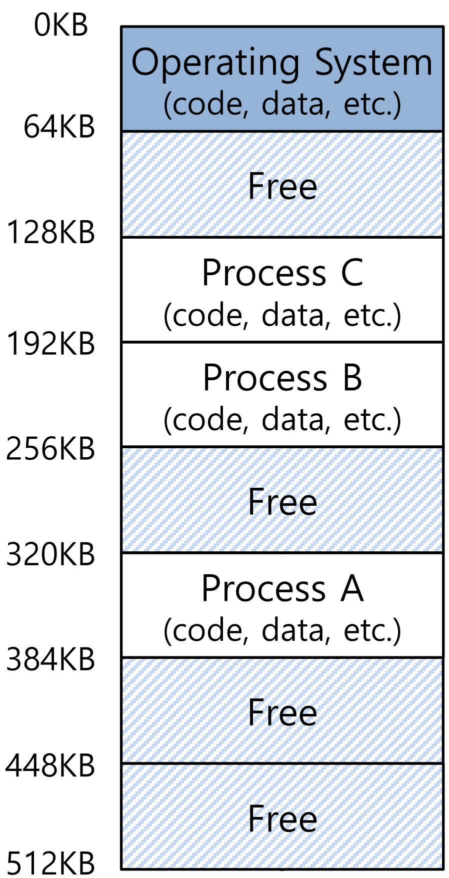
- Load only one process in memory.
  - Poor utilization and efficiency



Multiprogramming and Time Sharing

- **Load multiple processes** in memory.
  - Execute one for a short while.
  - Switch processes between them in memory.
  - Increase utilization and efficiency.

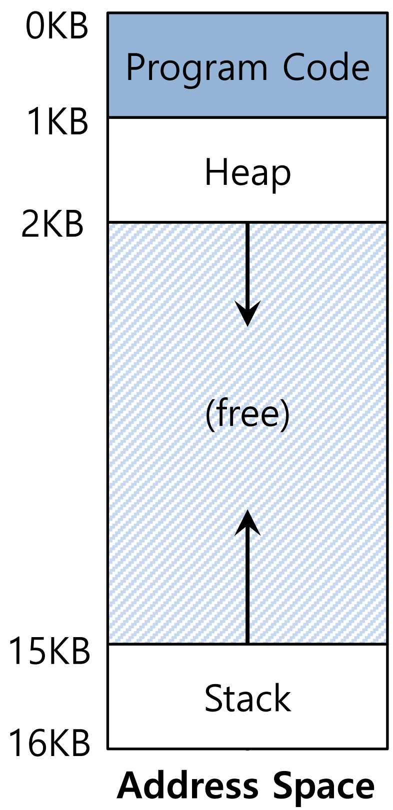
- Cause an important **protection issue**.
  - Errant memory accesses from other processes



Physical Memory

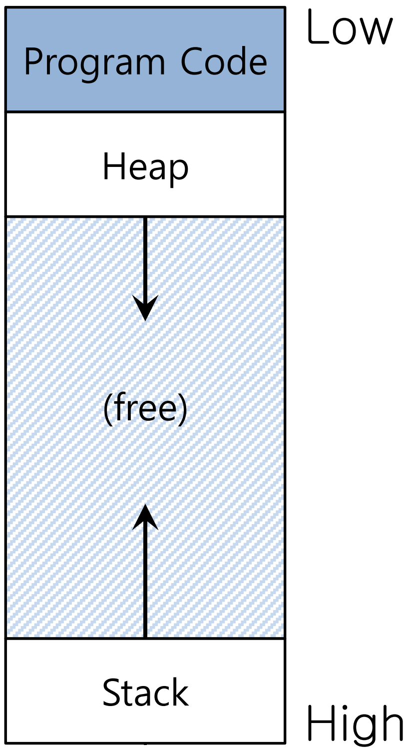
Virtual Address Space

- OS creates an **abstraction** of physical memory.
  - The address space contains all about a running process.
  - That consists of program code, heap, stack and etc.



Virtual Address Space(Cont.)

- Code
  - Where instructions live
- Heap
  - Dynamically allocate memory.
    - `malloc` in C language
    - `new` in object-oriented language
- Stack
  - Store return addresses or values.
  - Contain local variables arguments to routines.

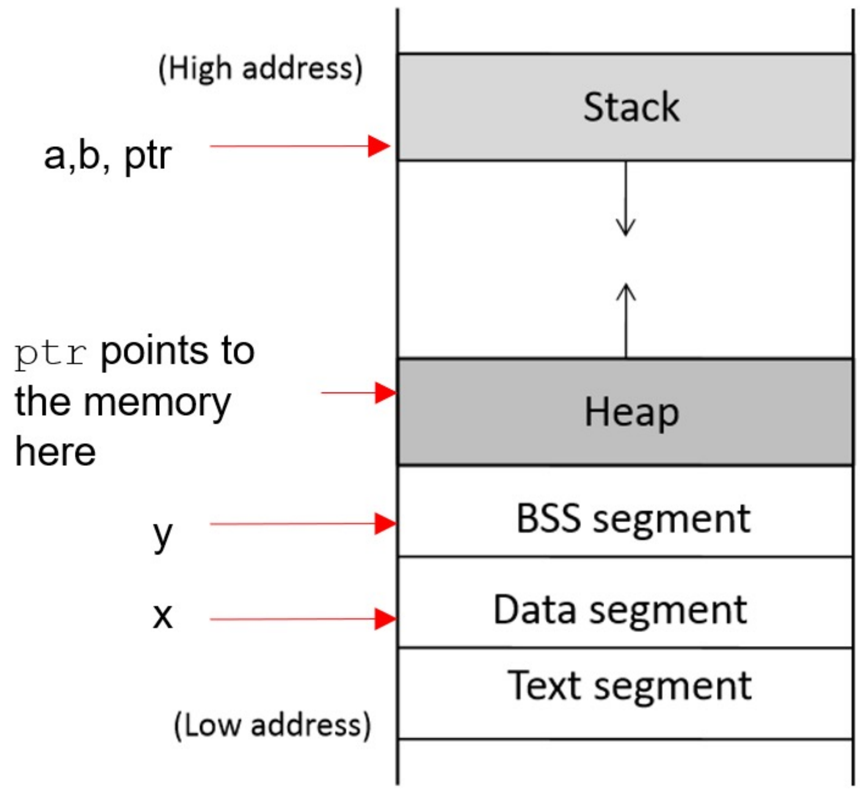


Address Space

Virtual Address Space(Cont.)

- Process memory layout

```
int x = 100;
int main()
{
   // data stored on stack
   int   a=2;
   float b=2.5;
   static int y;

   // allocate memory on heap
   int *ptr = (int *) malloc(2*sizeof(int));

   // values 5 and 6 stored on heap
   ptr[0]=5;
   ptr[1]=6;

   // deallocate memory on heap
   free(ptr);

   return 1;
}
```



Virtual Address

- **Every address** in a running program is virtual.

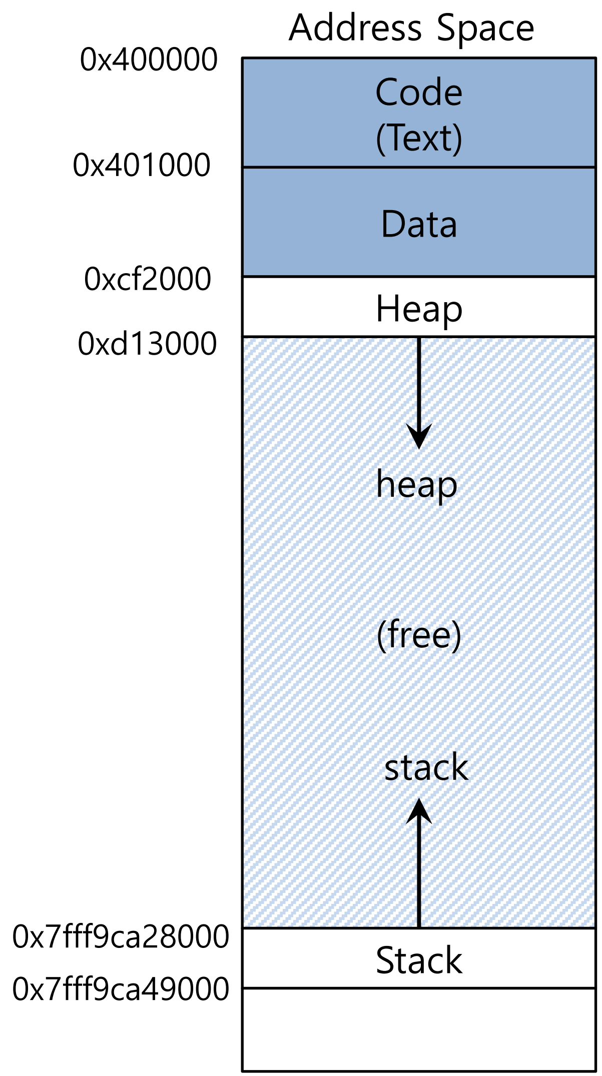
```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]){

    printf("location of code  : %p\n", (void *) main);
    printf("location of heap  : %p\n", (void *) malloc(1));
    int x = 3;
    printf("location of stack : %p\n", (void *) &x);

    return x;
}
```

A simple program that prints out addresses

Virtual Address(Cont.)

- The output in 64-bit Linux machine

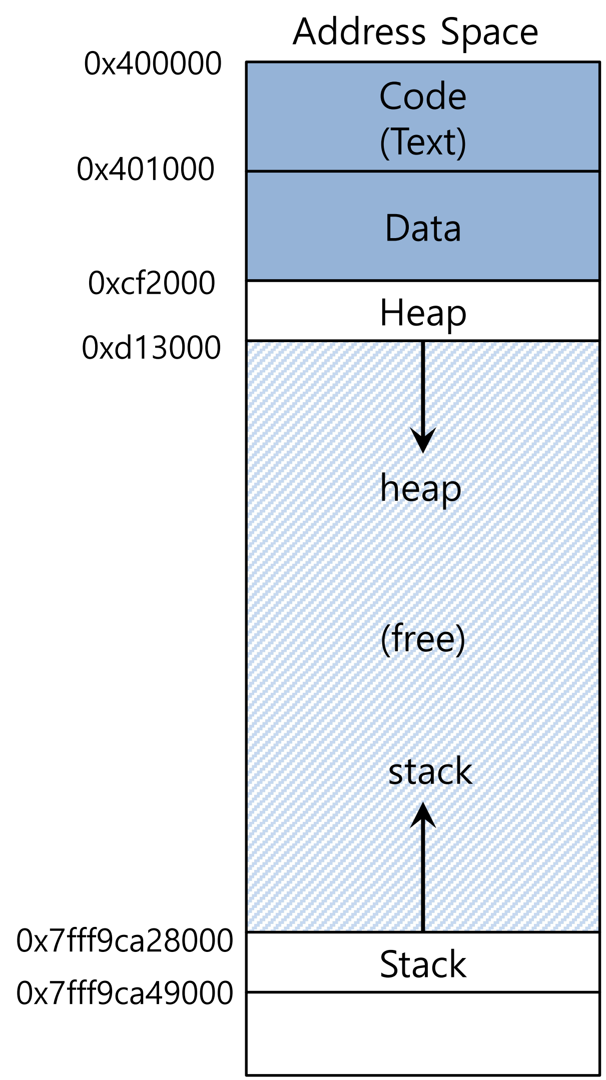
```
location of code  : 0x40057d
location of heap  : 0xcf2010
location of stack : 0x7fff9ca45fcc
```



Virtual Address(Cont.)

- The output in 64-bit Linux machine

```
location of code  : 0x40057d
location of heap  : 0xcf2010
location of stack : 0x7fff9ca45fcc
```



OS translates the virtual address to physical address

15. Address Translation

Address Translation

- Hardware transforms a **virtual address** to a **physical address**.
  - The desired information is actually located 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=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

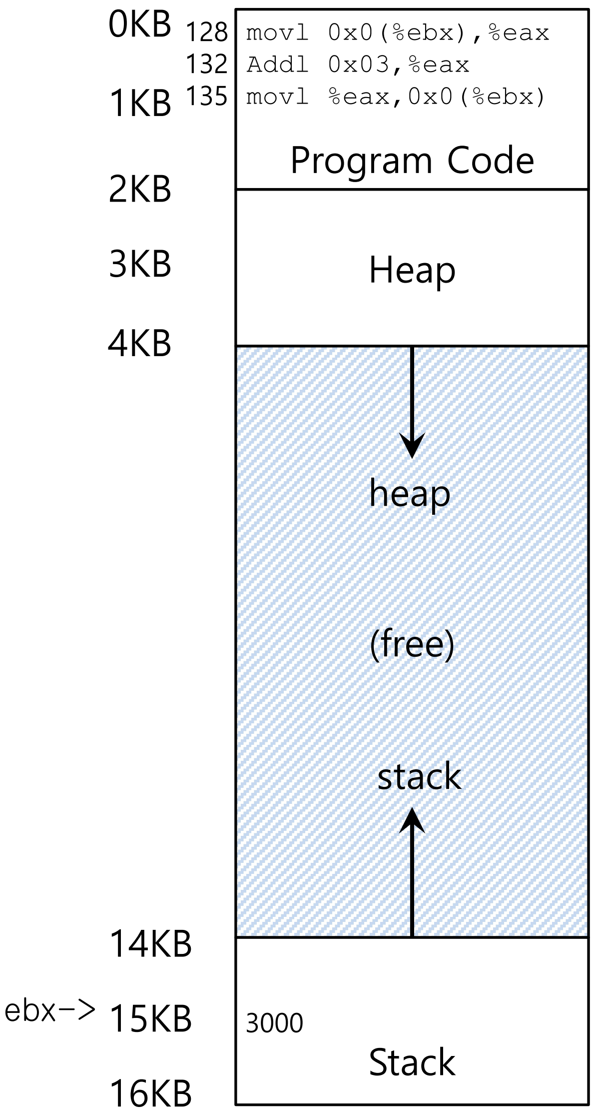
Example: Address Translation(Cont.)

- 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
```

- **Load** the value at that address into `eax` register.
- **Add** 3 to `eax` register.
- **Store** the value in `eax` back into memory.

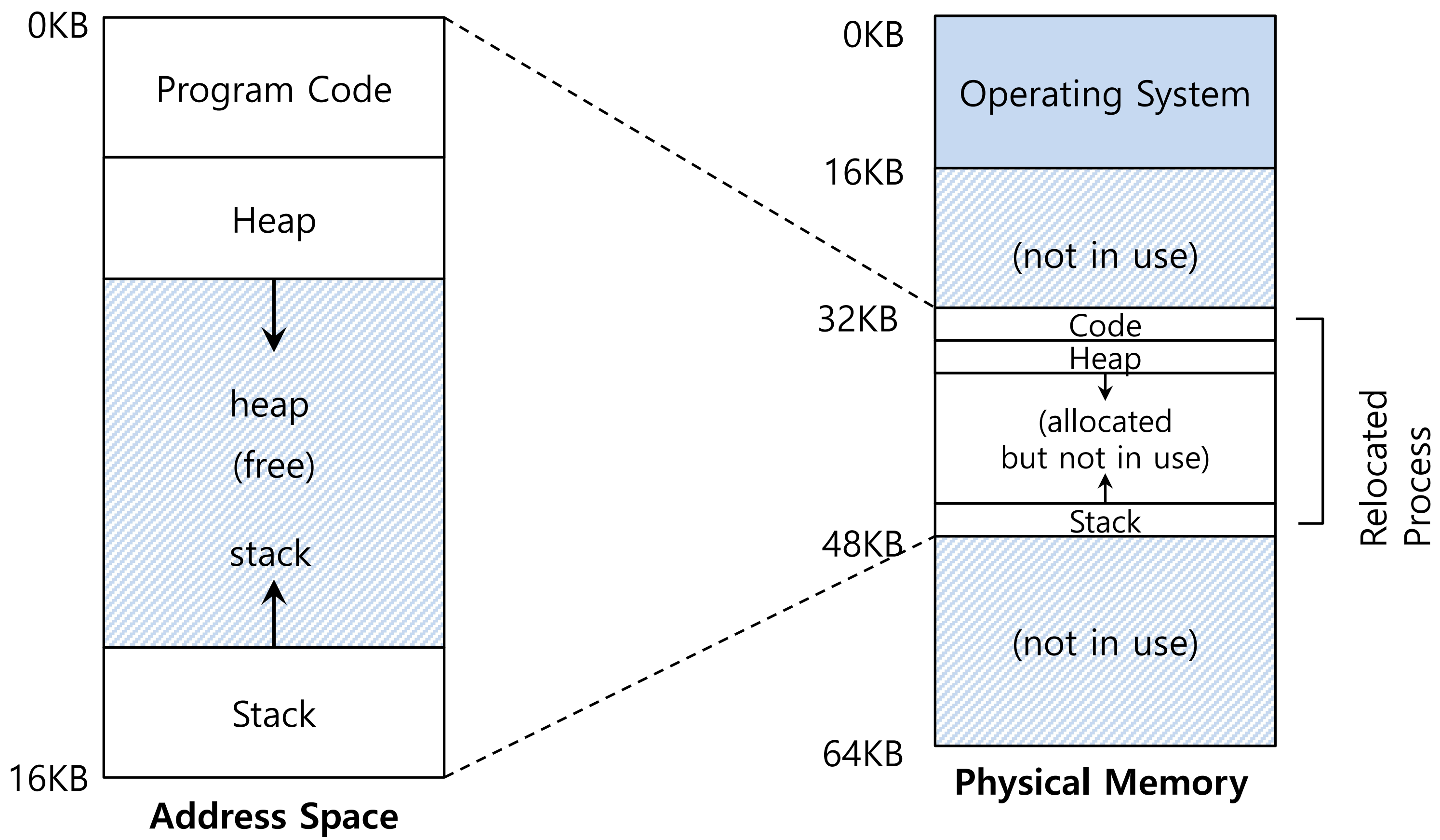
Example: Address Translation(Cont.)



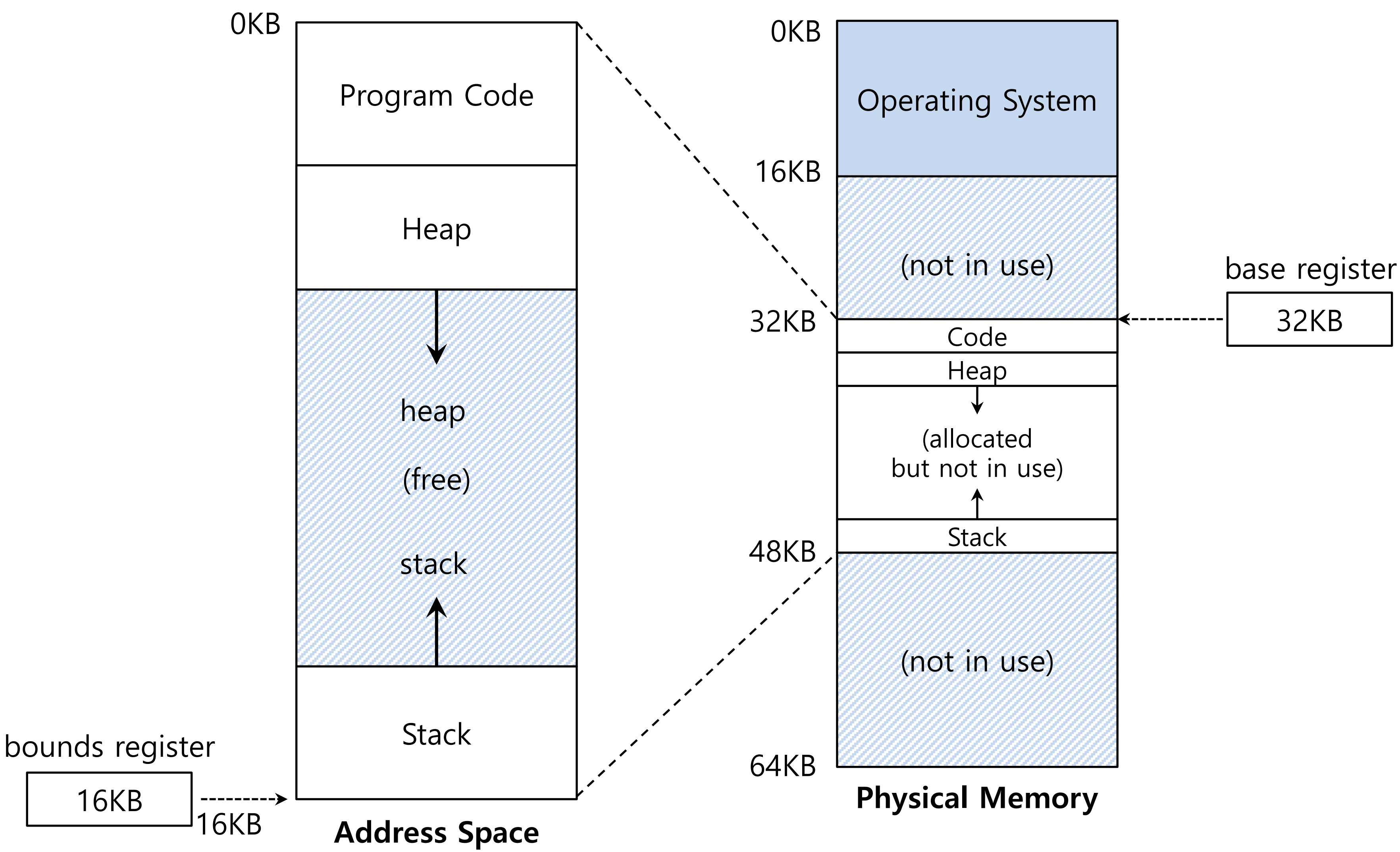
- Fetch instruction at address 128
- Execute this instruction (load from address 15KB)
- Fetch instruction at address 132
- Execute this instruction (no memory reference)
- Fetch the instruction at address 135
- Execute this instruction (store to address 15 KB)

Dynamic Relocation: Base and Bound Register

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



Base and Bounds Register



Dynamic(Hardware base) 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.**

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

Relocation and Address Translation

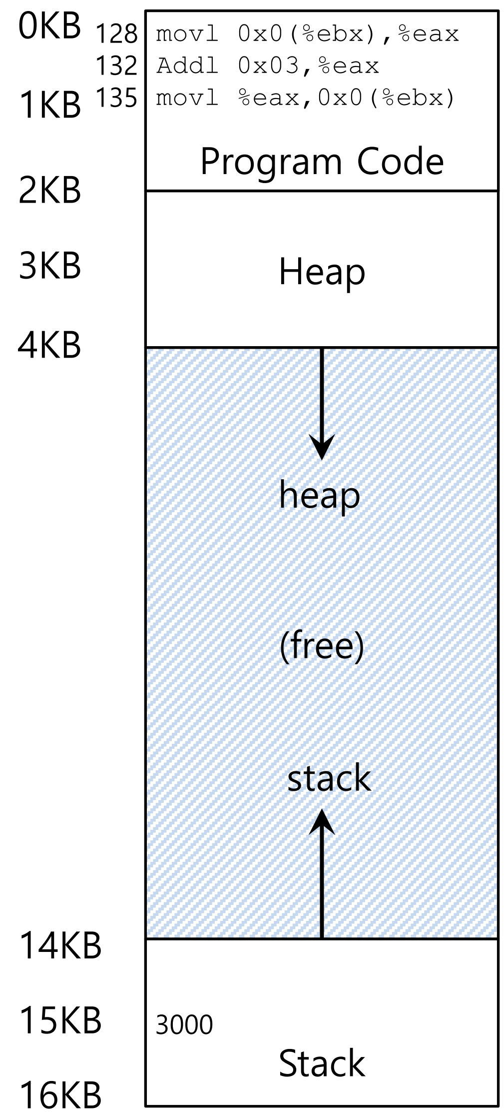
```
128 : movl 0x0(%ebx), %eax
```

- **Fetch** instruction at address 128

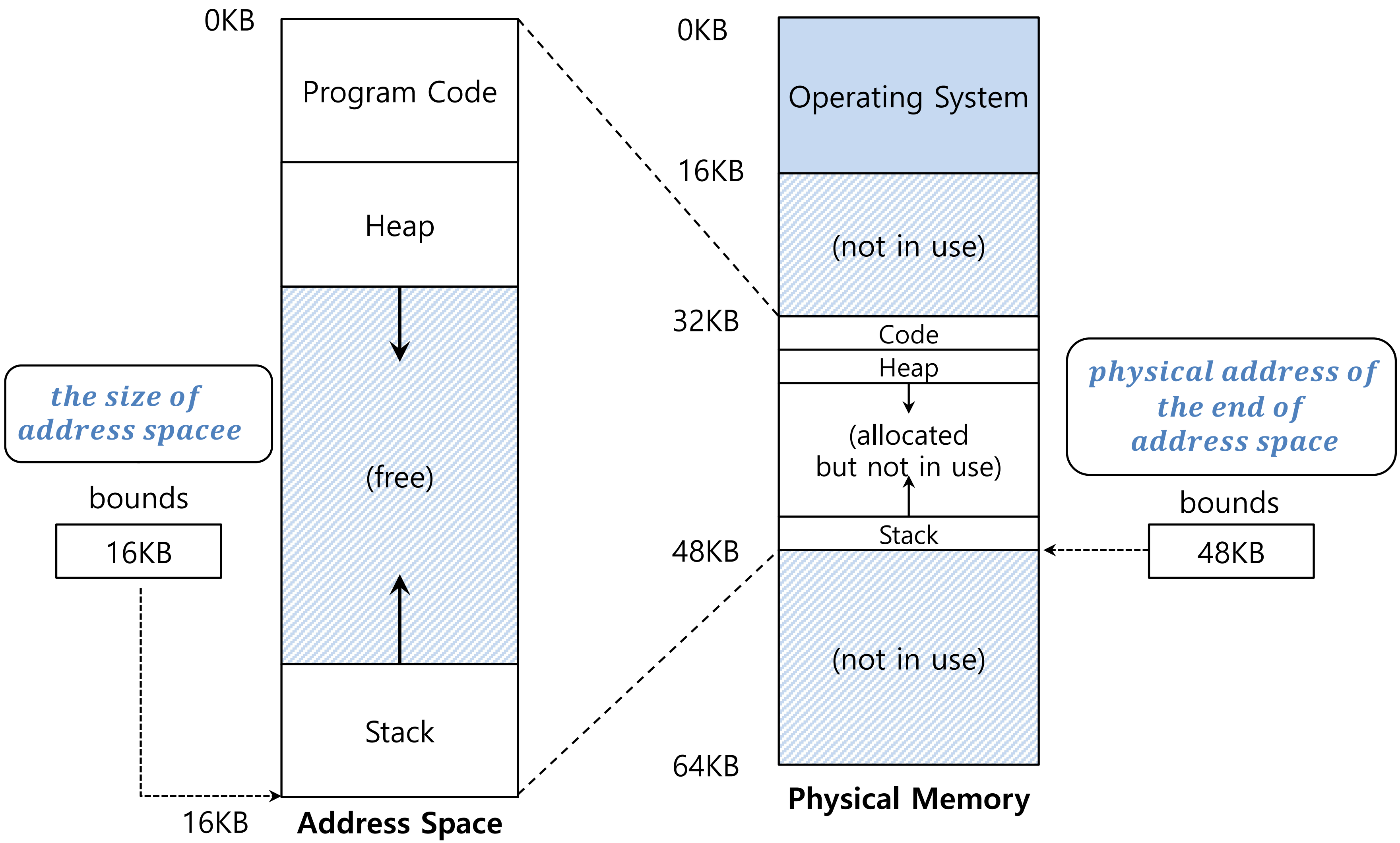
$$32896 = 128 + 32KB(base)$$

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

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



Bounds Register



Hardware Requirements

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*

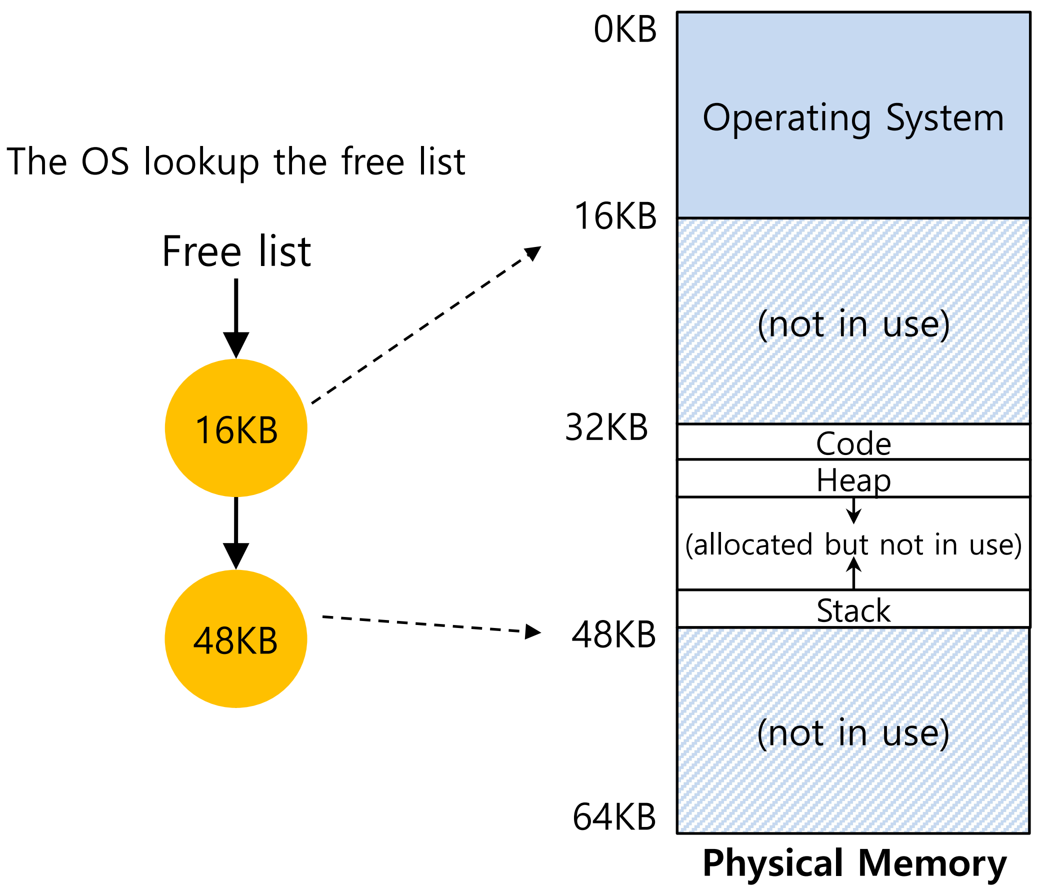
Dynamic Relocation: Hardware Requirements

OS Issues for Memory Virtualizing

- The OS must **take action** to implement **base-and-bounds** approach.
- Three critical actions:
  - 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

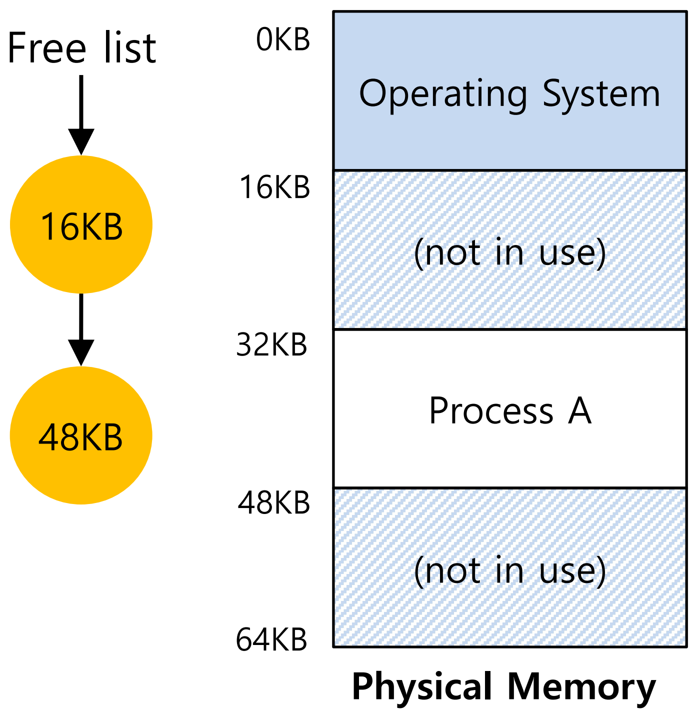
OS Issues: When a Process Starts Running

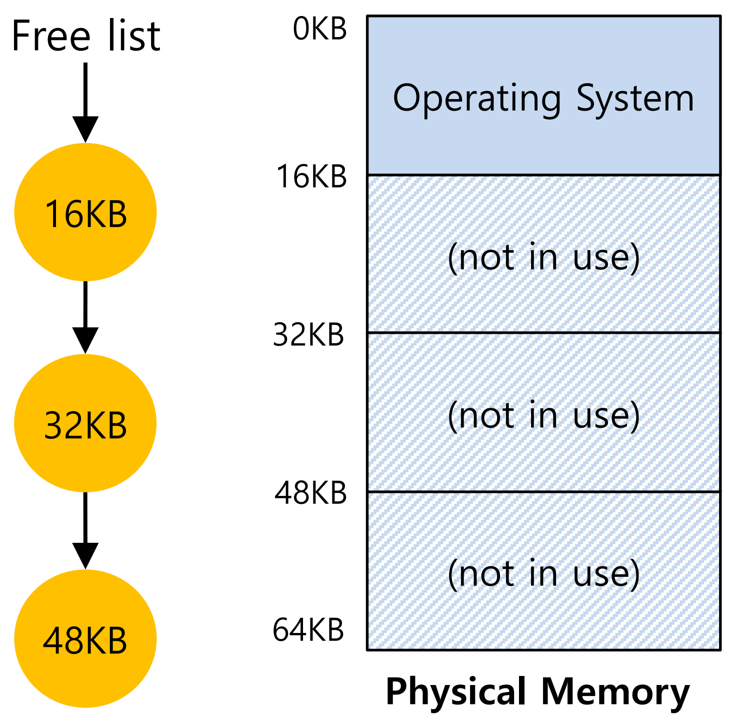
- The OS must **find a room** for a new address space.
  - free list : A list of the range of the physical memory which are not in use.



OS Issues: When a Process Is Terminated

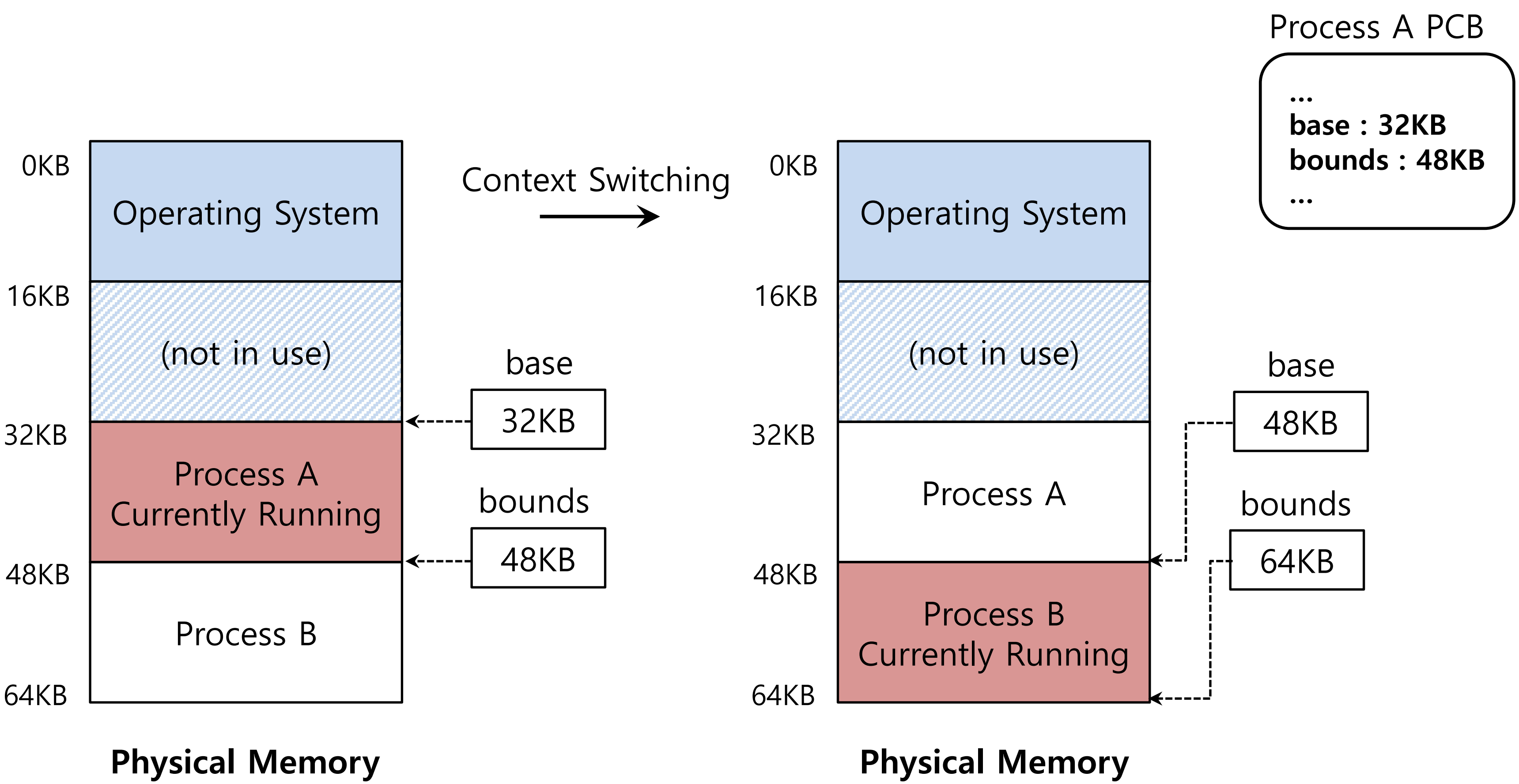
- The OS must **put the memory back** on the free list.





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)

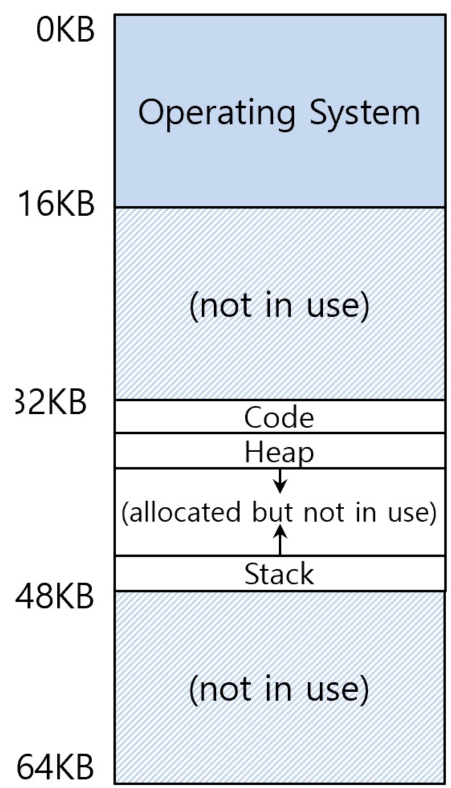


OS Issues: provide exception handlers

- the OS must provide exception handlers,
- the OS installs these handlers at boot time (via privileged instructions
  - Exception handler for segmentation fault

Inefficiency of Base and Bound registers

- Internal fragmentation
  - The relocated process is using physical memory from 32 KB to 48 KB
  - The process stack and heap are not too big, all of the space between the two is simply *wasted*.



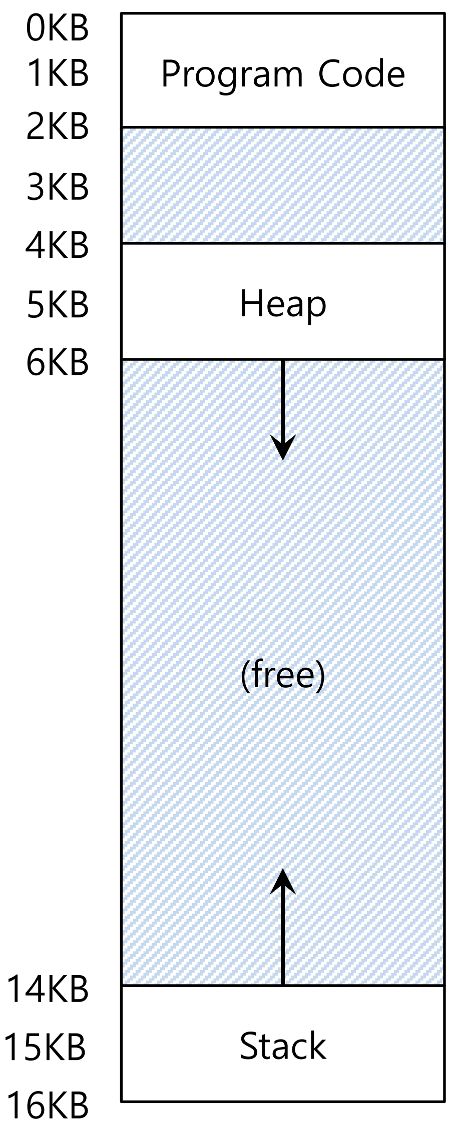
Physical Memory

Summary

- Address translation: hardware support and OS support
- Basic form: base and bound
- Fragmentation issue

16. Segmentation

Inefficiency of the Base and Bound Approach



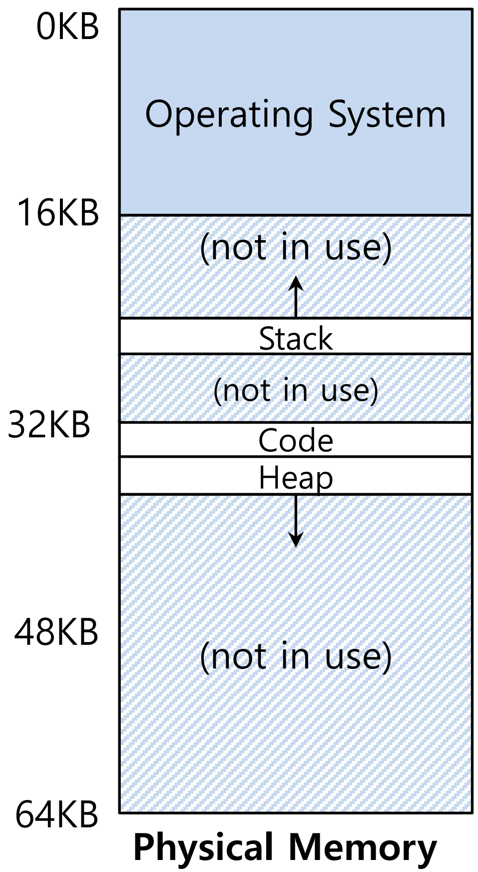
- **Big chunk of "free"** space
- "free" space **takes up** physical memory.
- Hard to run when an address space **does not fit** into physical memory

Segmentation

- Segment is just **a contiguous portion** of the address space of a particular length.
  - Logically-different segment: code, stack, heap

- Each segment can be **placed** in **different part of physical memory**.
  - **Base** and **bounds** exist **per each segment**.

Placing Segment In Physical Memory



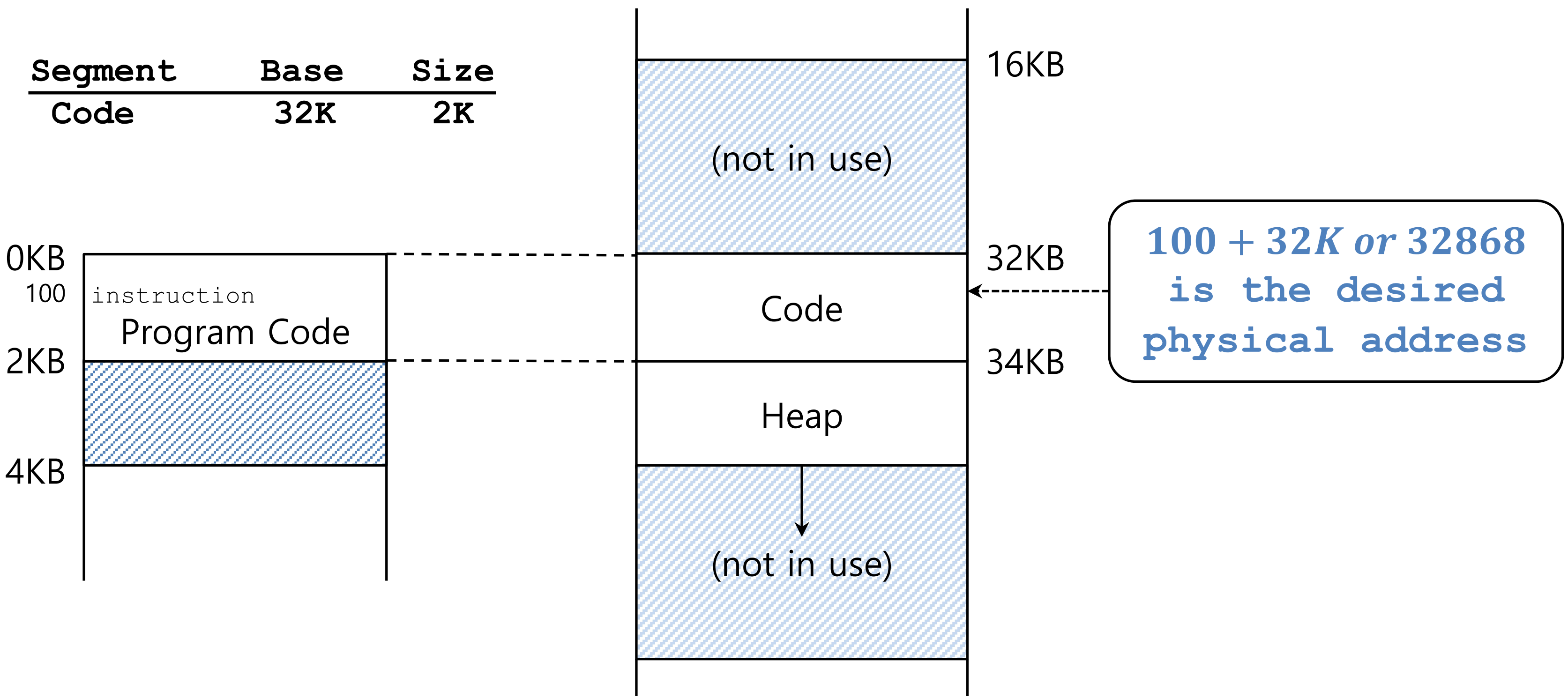
Segment	Base	Size
Code	32K	2K
Heap	34K	2K
Stack	28K	2K

Address Translation on Segmentation: code

$$physical\ address = offset + base$$

- The `offset` of virtual address 100 is `100`.
  - The code segment **starts at virtual address 0** in address space.

Segment	Base	Size
Code	32K	2K

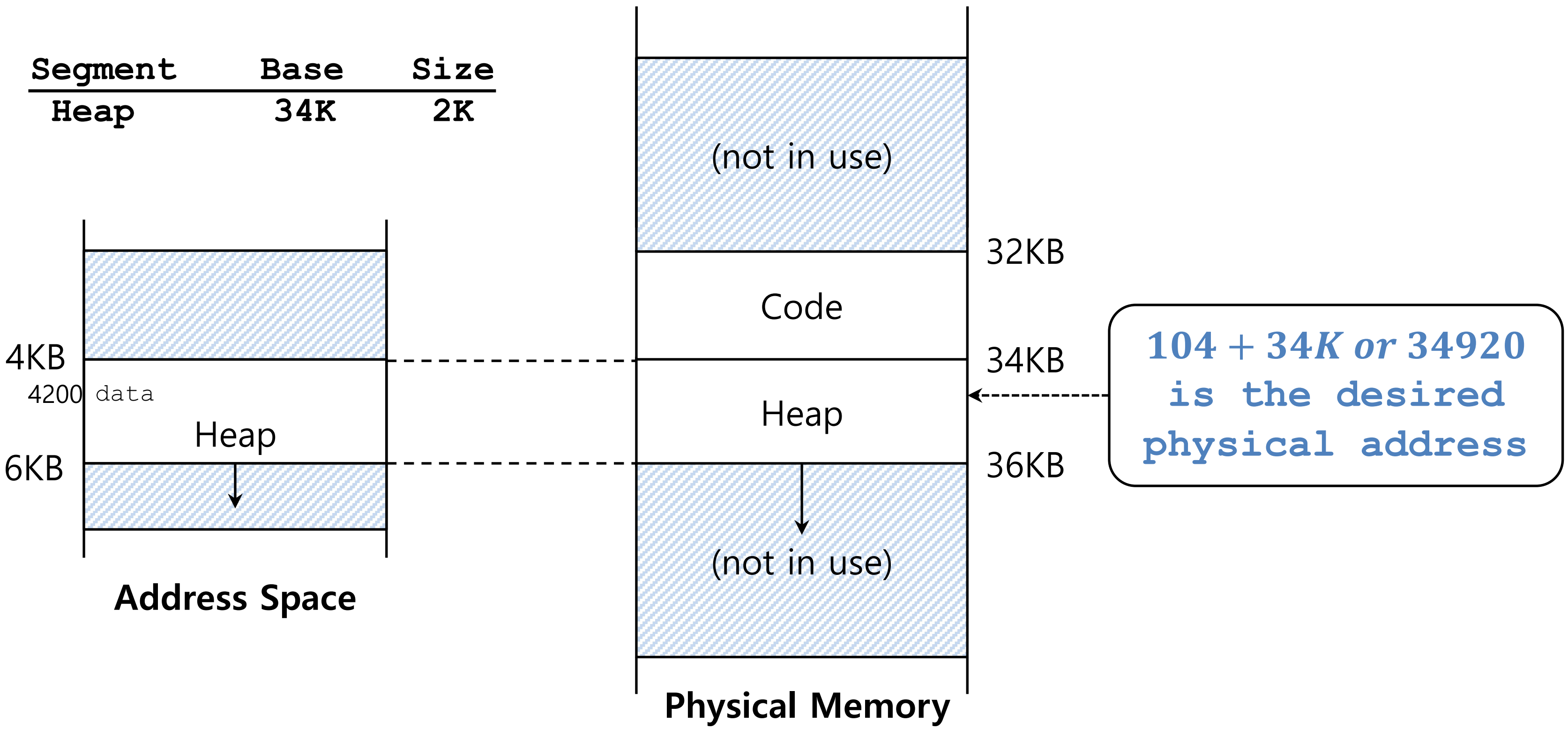


Address Translation on Segmentation: heap

Virtual address + *base* is not the correct physical address.
OFFSET of *Virtual address* + *base* is the correct physical address.

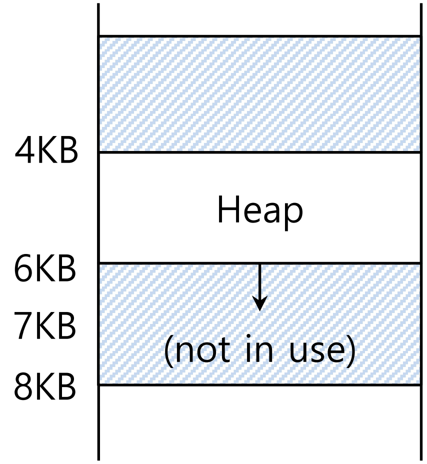
- The `offset` of virtual address 4200 is `104`.
  - The heap segment **starts at virtual address 4096** in address space.

Segment	Base	Size
Heap	34K	2K



Segmentation Fault or Violation

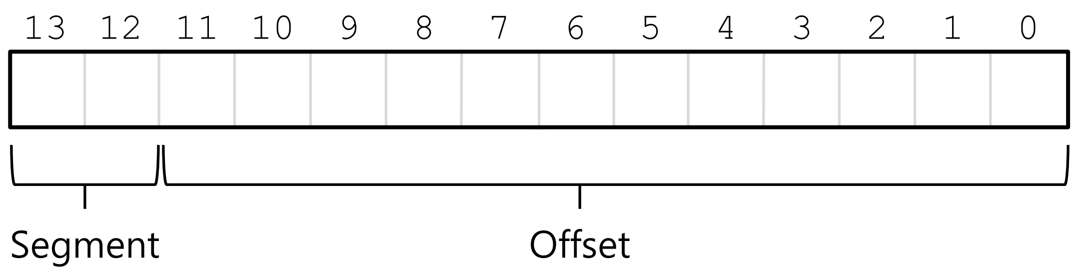
- If an **illegal address** such as 7KB which is beyond the end of heap is referenced, the OS occurs **segmentation fault**.
  - The hardware detects that address is **out of bounds**.



Address Space

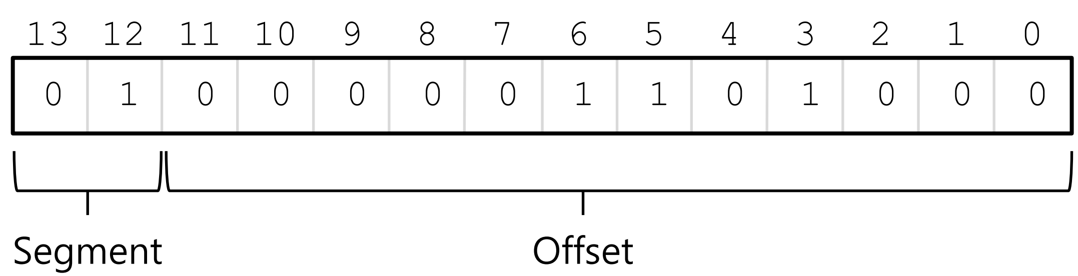
Referring to Segment

- **Explicit approach**
  - Chop up the address space into segments based on the **top few bits** of virtual address.



- Example: virtual address 4200 (01000001101000)

Segment	bits
Code	00
Heap	01
Stack	10
-	11



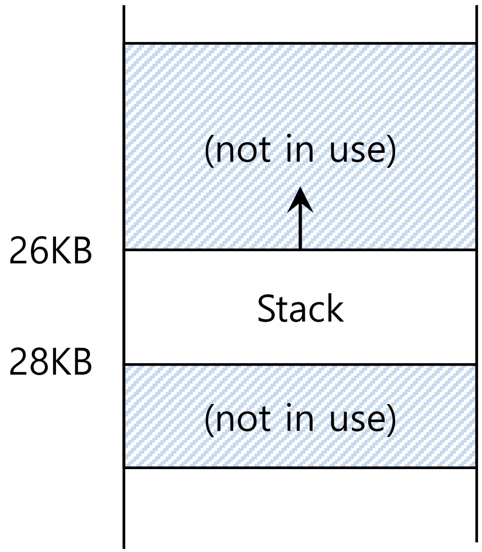
Segment selection

```
// get top 2 bits of 14-bit VA
Segment = (VirtualAddress & SEG_MASK) >> SEG_SHIFT
// now get offset
Offset = VirtualAddress & OFFSET_MASK
if (Offset >= Bounds[Segment])
     RaiseException(PROTECTION_FAULT)
else
     PhysAddr = Base[Segment] + Offset
     Register = AccessMemory(PhysAddr)
```

- SEG_MASK = 0x3000(11000000000000)
- SEG_SHIFT = 12
- OFFSET_MASK = 0xFFF (00111111111111)

Referring to Stack Segment

- Stack grows **backward**.
- **Extra hardware support** is need.
  - The hardware checks which way the segment grows.
  - 1: positive direction, 0: negative direction



Physical Memory

Segment Register(with Negative-Growth Support)

Segment	Base	Size	Grows Positive?
Code	32K	2K	1
Heap	34K	2K	1
Stack	28K	2K	0

Support for Sharing

- Segment can be **shared between address** space.
  - **Code sharing** is still in use in systems today.
  - by extra hardware support.
- Extra hardware support is need for form of **Protection bits.**
  - **A few more bits** per segment to indicate **permissions** of **read,** write and **execute**.

Segment Register Values(with Protection)

Segment	Base	Size	Grows Positive?	Protection
Code	32K	2K	1	Read-Execute
Heap	34K	2K	1	Read-Write
Stack	28K	2K	0	Read-Write

Fine-Grained and Coarse-Grained segmentation

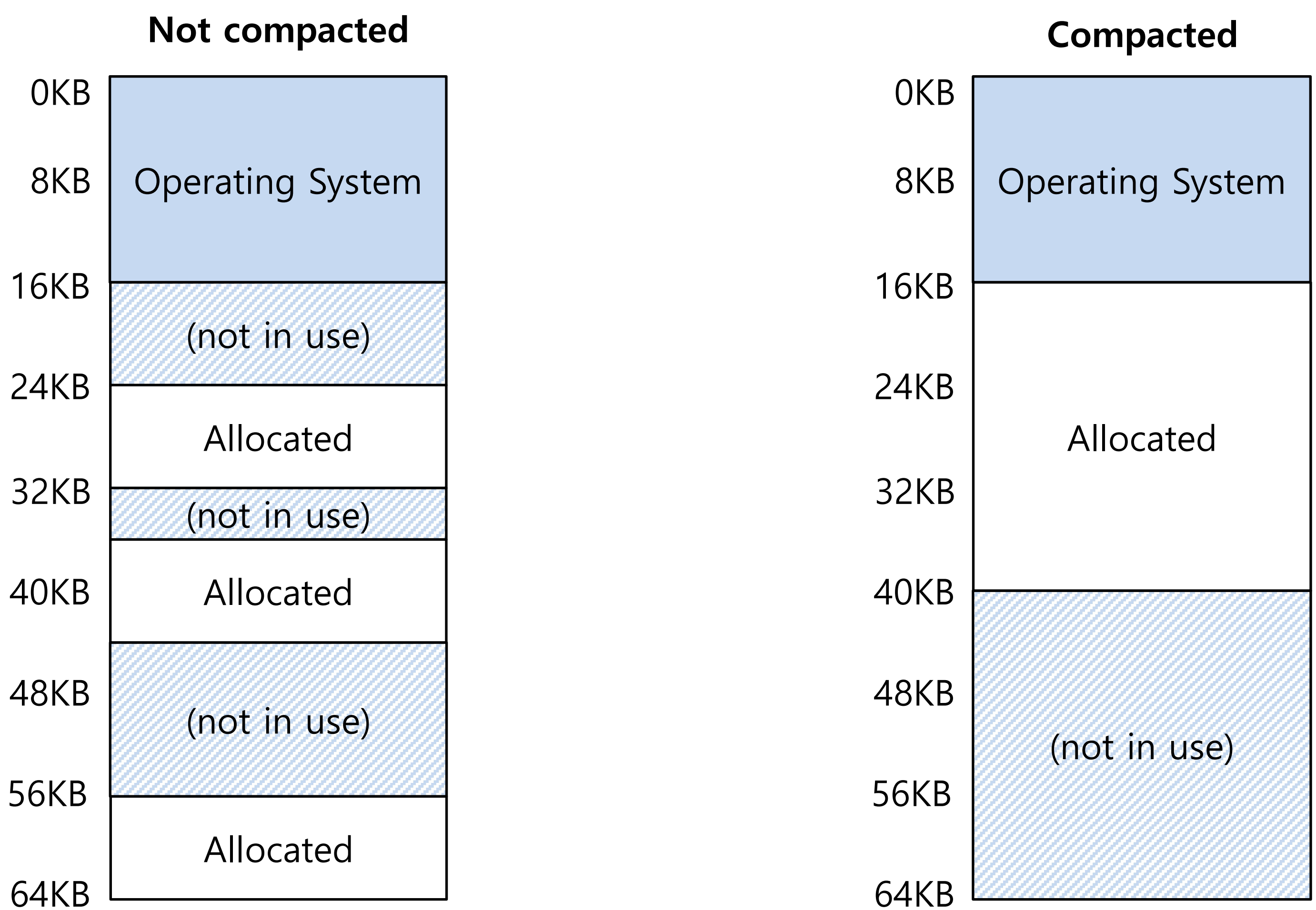
- **Coarse-Grained** means small number of segments.
  - e.g., code, heap, stack.
- **Fine-Grained** segmentation allows **more flexibility** for address space in some early system.
  - To support many segments, Hardware support with a **segment table** is required.

OS support: Fragmentation

- **External Fragmentation**: little holes of **free space** in physical memory that is too small for allocating segment.
  - There is **24KB free**, but **not in one contiguous** segment.
  - The OS **cannot** satisfy the **20KB request**.

- **Compaction**: **rearranging** the exiting segments in physical memory.
  - Compaction is **costly**.
    - **Stop** running process.
    - **Copy** data to somewhere.
    - **Change** segment register value.

Memory Compaction



History of segmentation

- In early days, OS used segmentation.
  - Burroughs B5000 (first commercial machine with virtual memory)
  - IBM AS/400
  - Intel 8086, 80286
- 80386 and later Intel CPU’s support paging.
- x86-64 does not use segmentation any more in 64bit mode
  - CS,SS,DS and ES are forced to 0 and 2^24..

Summary

- Segmentation can better support sparse address spaces.
- It is also fast as the overheads of translation are minimal.
- Sharing (such as code) is easy.
- Issues
  - External fragmentation issue
  - Sparse segment