CSCI3150 Introduction to Operating Systems

Lecture 10: Memory Management I

Hong Xu

https://github.com/henryhxu/CSCI3150

Overview

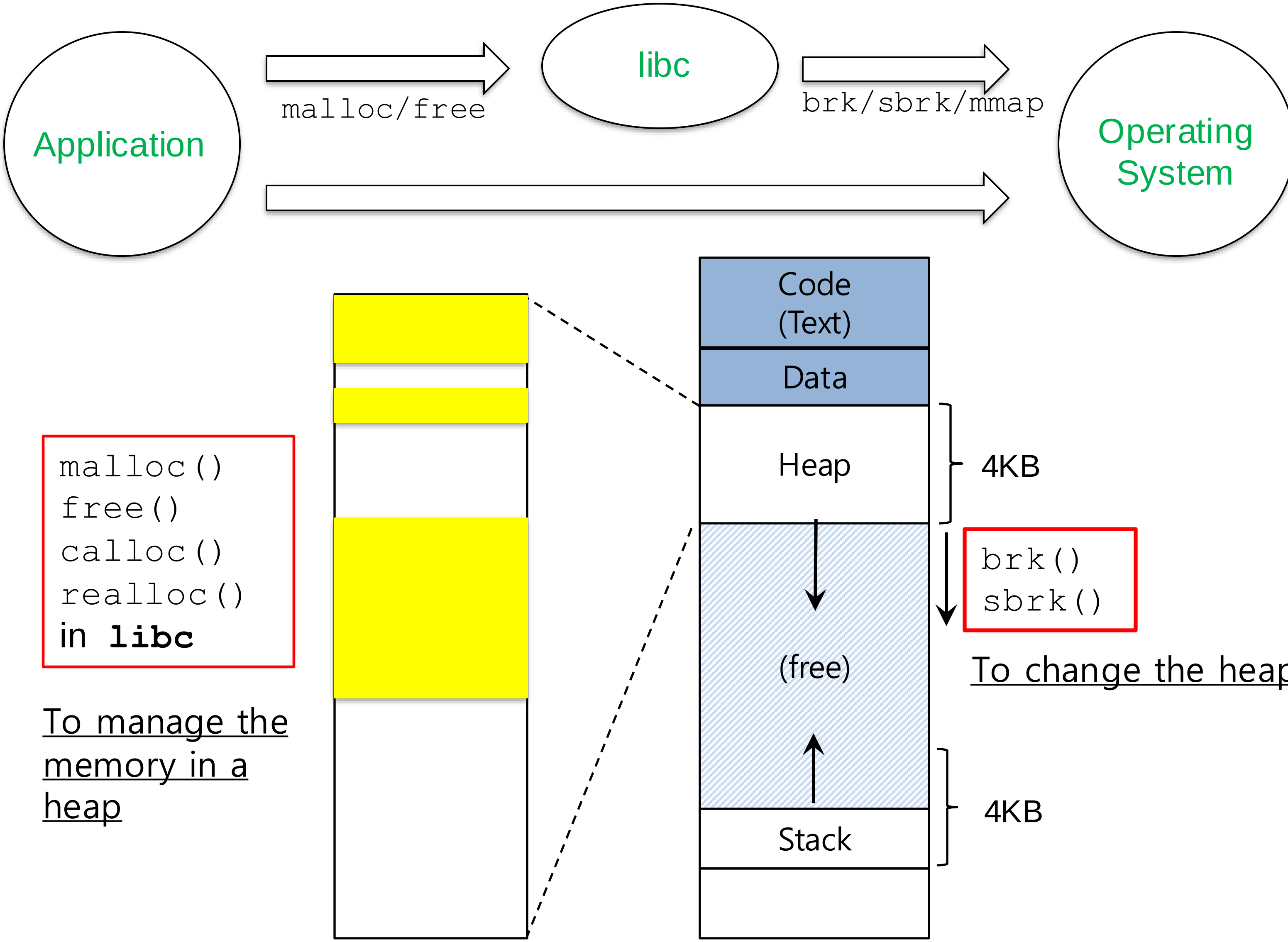
- Address space, memory API
- Address translation: Base-and-bounds
- Segmentation

Memory API

Overview

- `malloc/free`
- `calloc/realloc`
- `brk/sbrk`

Virtual Address Space




```
malloc()
free()
calloc()
realloc()
```
in **libc**

To manage the memory in a heap

```
brk()
sbrk()
```

To change the heap size

malloc()

```
#include <stdlib.h>

void* malloc(size_t size)
```

- Allocate a memory region on the heap.
  - Argument
    - `size_t size` : size of the memory block (in bytes)
    - `size_t` is an unsigned integer type
  - Return
    - Success: a void type pointer to the memory block allocated by `malloc`
    - Fail: a null pointer

sizeof()

- Routines and macros are utilized for `size` in `malloc` instead typing in a number directly.
- Two types of results when using `sizeof` on variables; be careful…
  - The actual size of `'x'` is known at run-time.

```
int *x = malloc(10 * sizeof(int));
printf("%d\n", sizeof(x));
```

```
4
```

  - The actual size of `'x'` is known at compile-time.

```
int x[10];
printf("%d\n", sizeof(x));
```

```
40
```

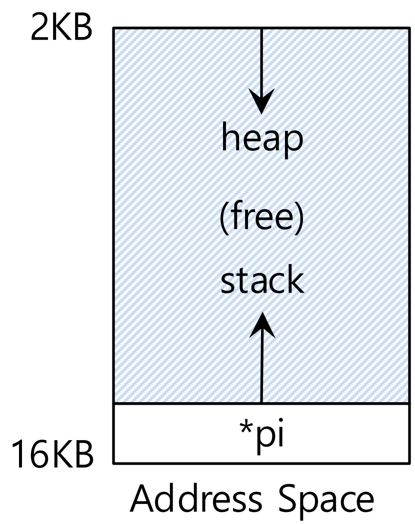
Memory API: `free()`

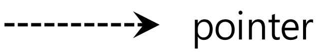
```
#include <stdlib.h>

void free(void* ptr)
```

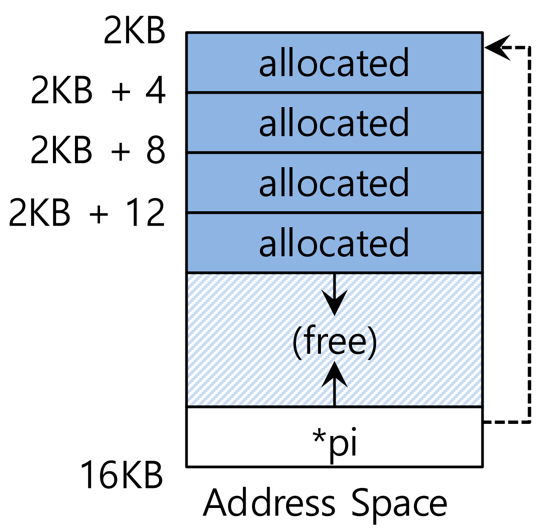
- Free a memory region allocated by a call to `malloc`
  - Argument
    - `void *ptr`: a pointer to a memory block allocated with `malloc`
  - Return
    - none

Memory Allocating



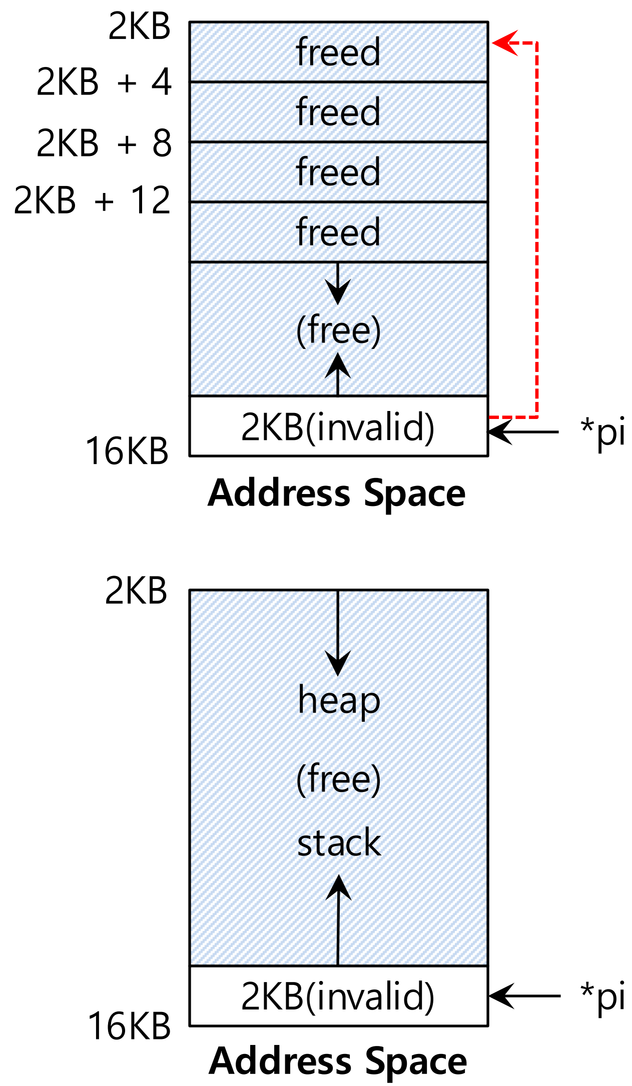



```
int *pi; // local variable
```




```
pi = (int *)malloc(sizeof(int)* 4);
```

Memory Freeing



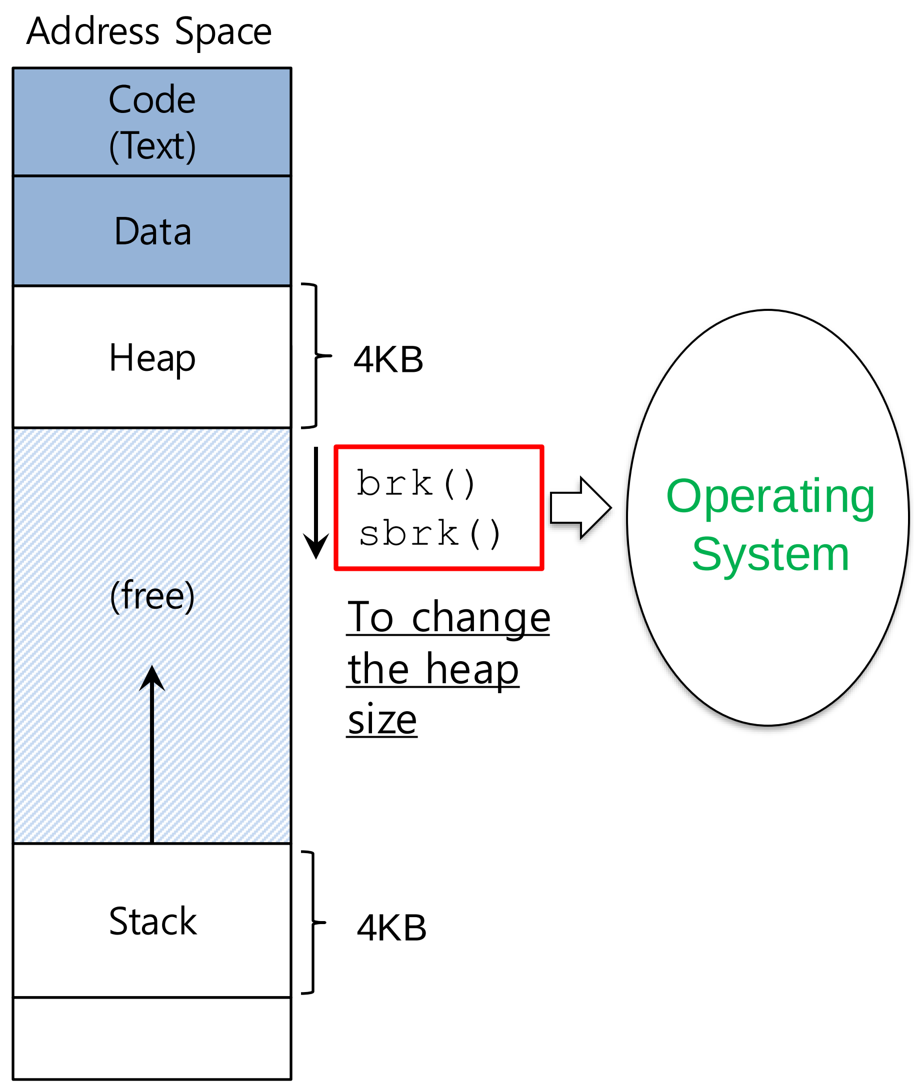

```
free(pi);
```

System Calls

```
#include <unistd.h>

int brk(void *addr)
void *sbrk(intptr_t increment);
```

- There lacks heap space → Ask OS to expand the heap
- `break`: The location of **the end of the heap** in address space
- `malloc` uses `brk` syscall
    - `brk` is called to expand the program's *break*.
    - `sbrk` is similar to `brk`.
    - Programmers **should never directly call** either `brk` or `sbrk`.



Address Translation

Memory Virtualization with Efficiency and Control

- Memory virtualization takes a similar strategy known as **limited direct execution (LDE)**
- Efficiency and control are attained by hardware support.
  - e.g., registers, TLBs (Translation Look-aside Buffers), page table

Address Translation

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

- Code in C

```
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

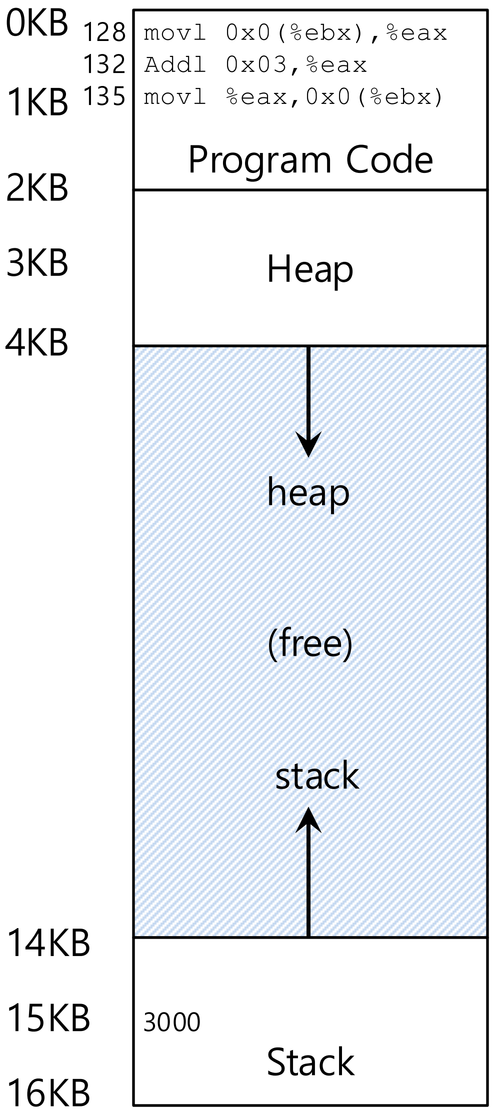
Example: Address Translation (Cont.)

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

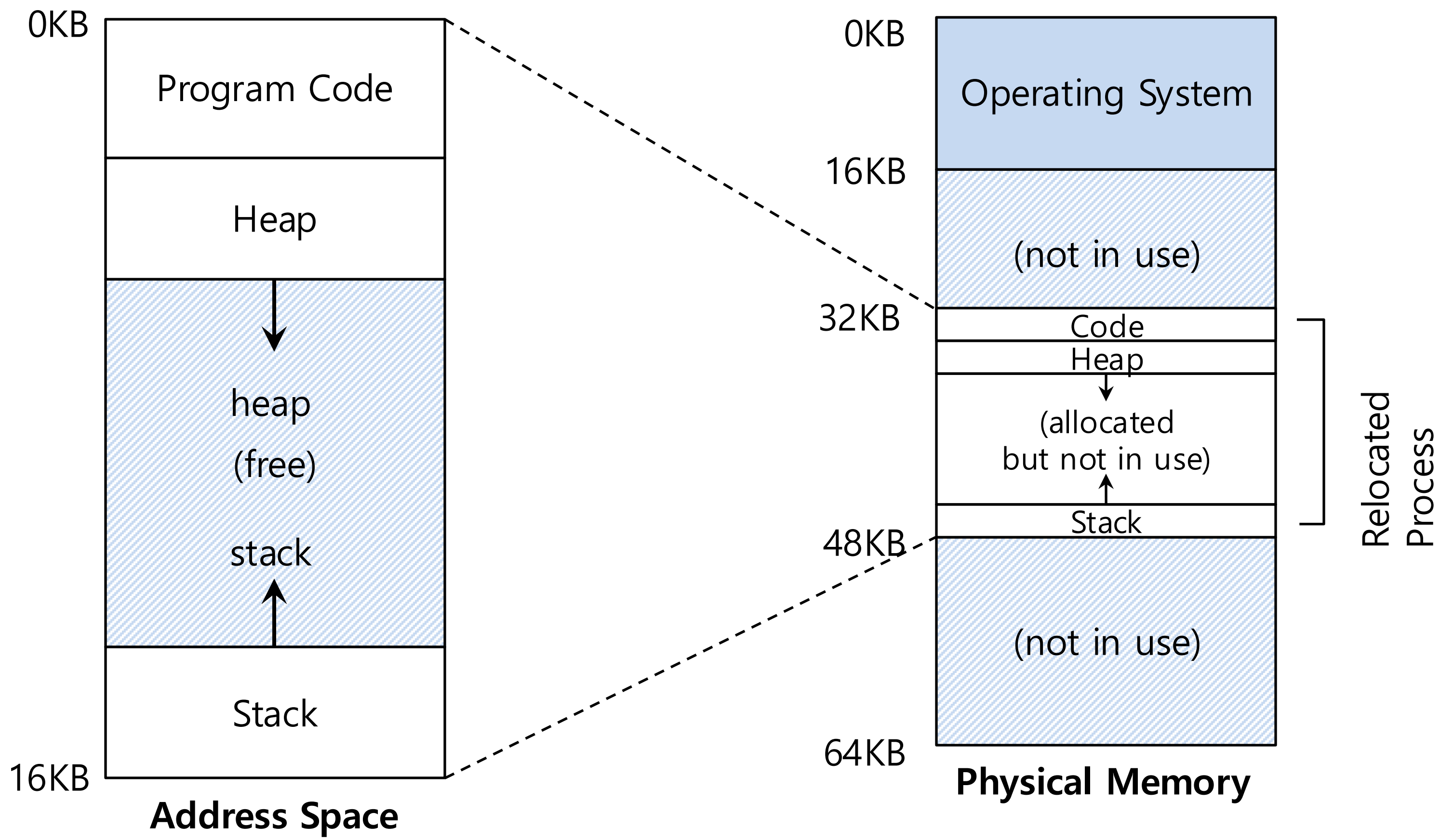
Example: Address Translation (Cont.)



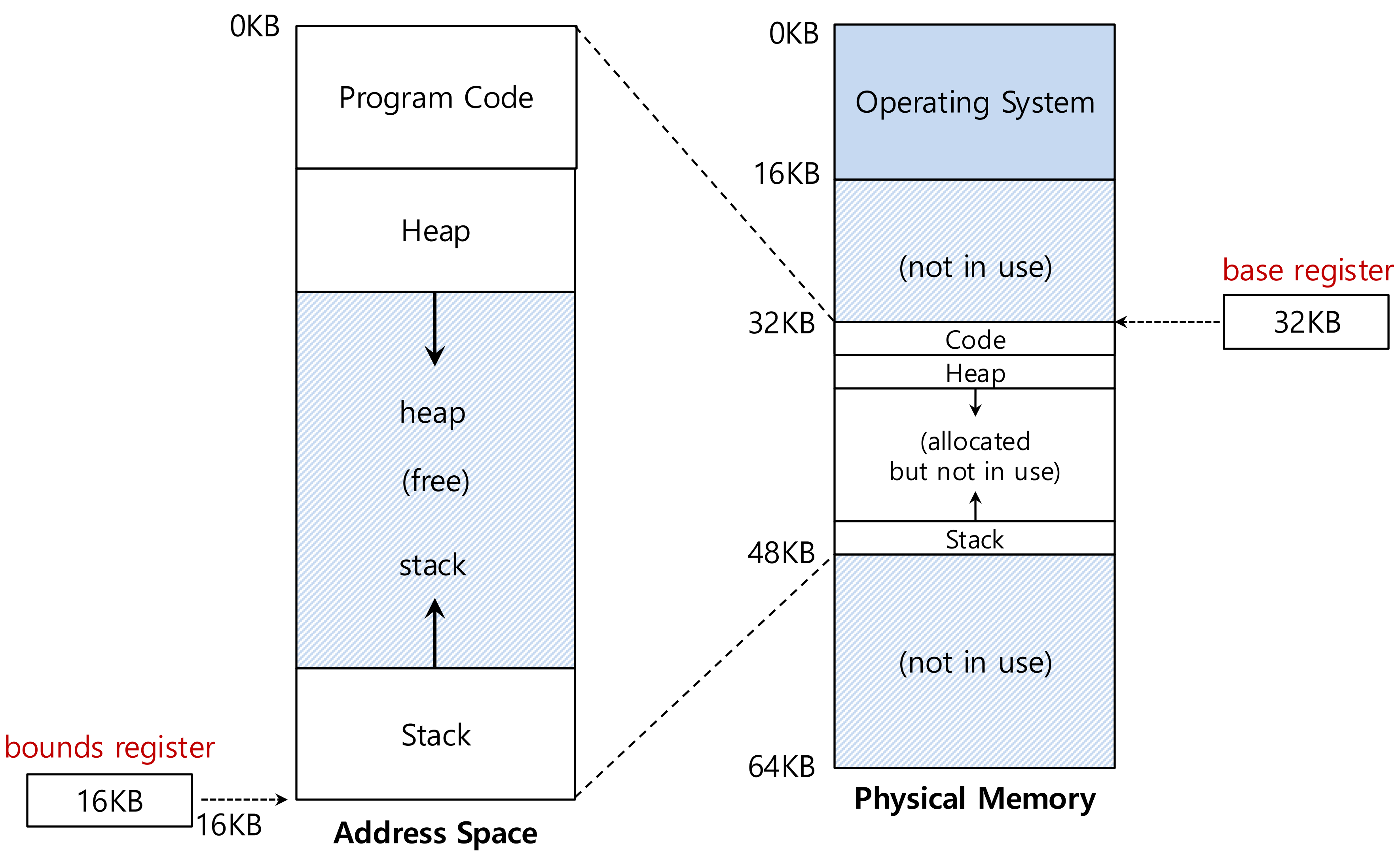
- Fetch instruction at address 128
- Execute this instruction (load from address 15KB)
- Fetch instruction at address 132
- Execute this instruction (no memory reference)
- Fetch the instruction at address 135
- Execute this instruction (store to address 15 KB)

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

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



Base and Bounds Register



Base and Bounds

- 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 bounds, or negative**

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

Relocation and Address Translation

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

- **Fetch** instruction at address 128

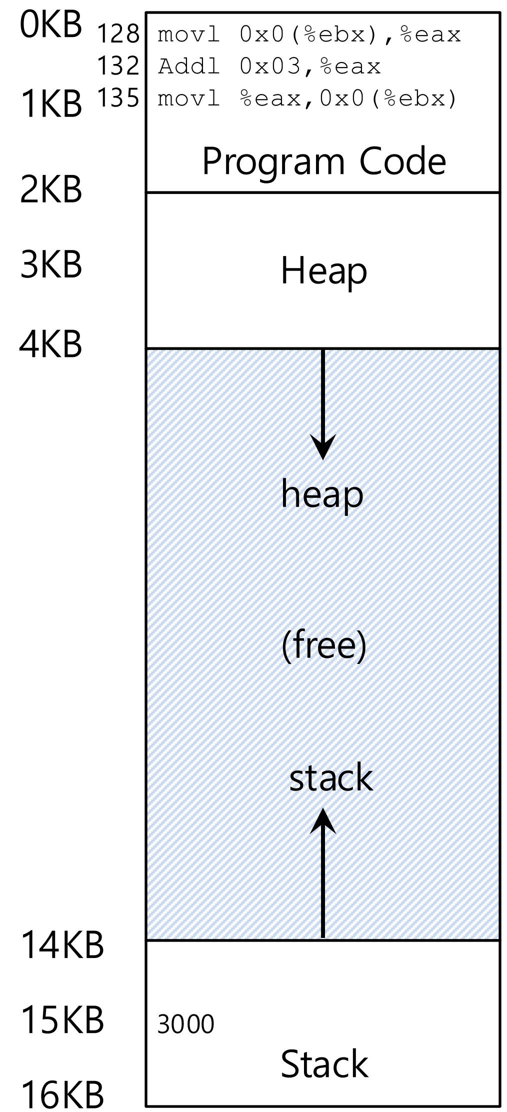
$$32896 = 128 + 32KB(base)$$

- **Execute** this instruction

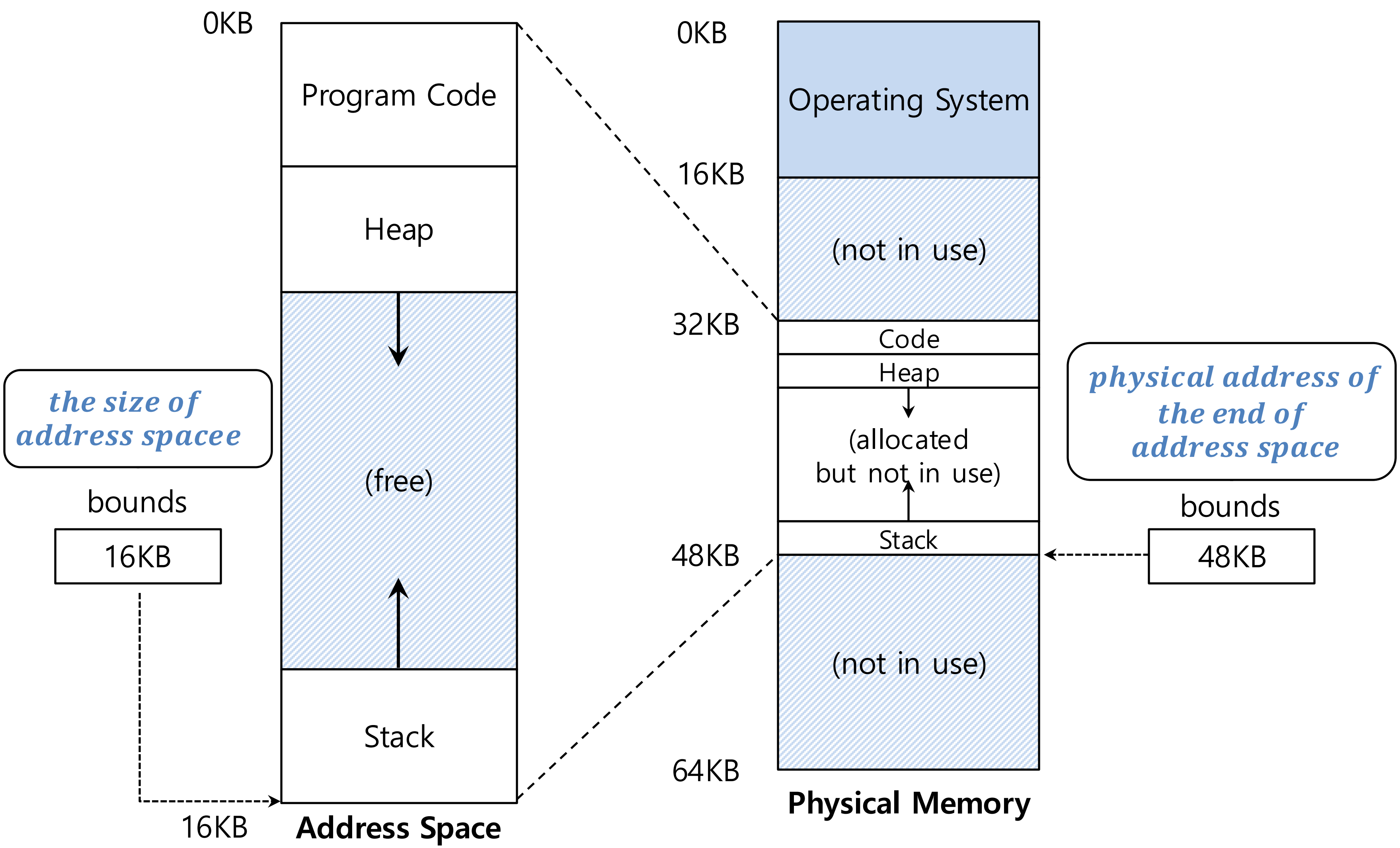
$$1\text{KB} = 1024\text{B}$$

  - Load from address 15KB

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



Two Ways of Bounds Register



Hardware Requirements

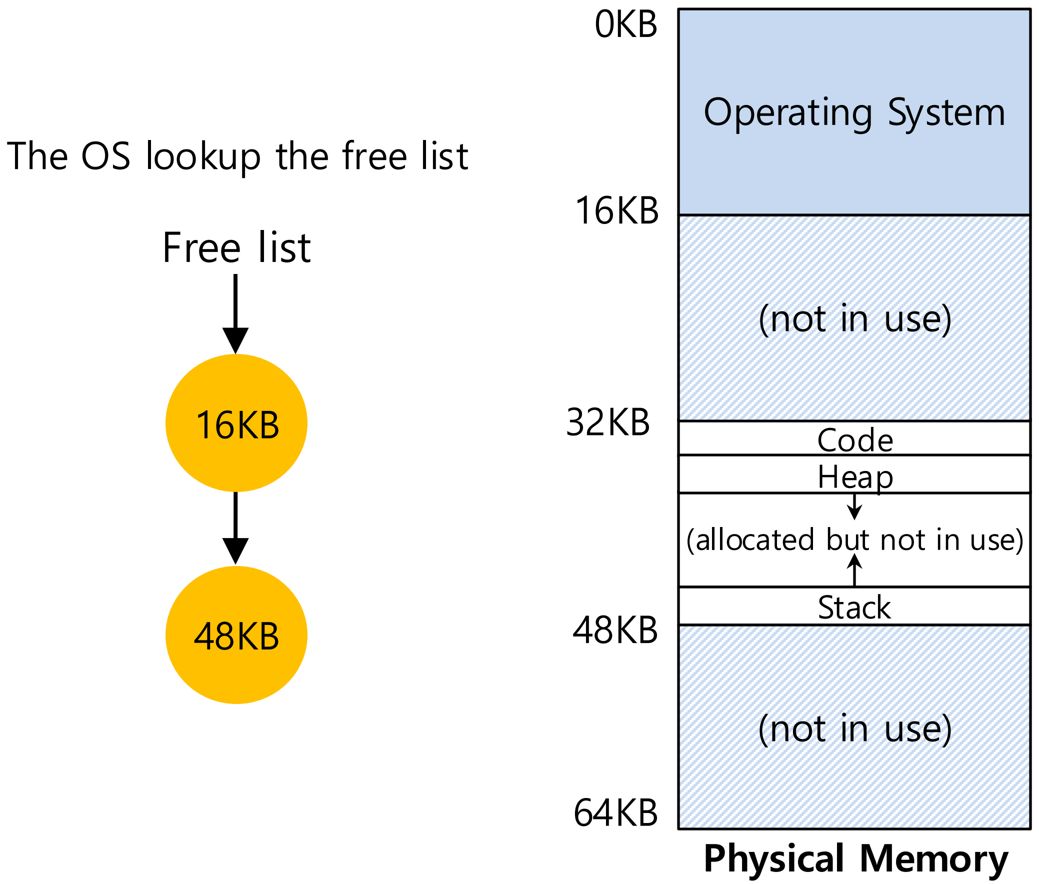
- Privileged mode: prevent user-mode processes from executing privileged operations
- Base/bounds registers: Need a pair of registers per CPU to support address translation and bounds checks
- Ability to translate virtual addresses and check if within bounds limits; Circuitry to do translations.
- Privileged 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: OS must be able to tell hardware what exception handlers code to run if exception occurs
- Ability to raise exceptions when processes try to access privileged instructions or out-of-bounds memory

OS Issues for Memory Virtualization

- 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

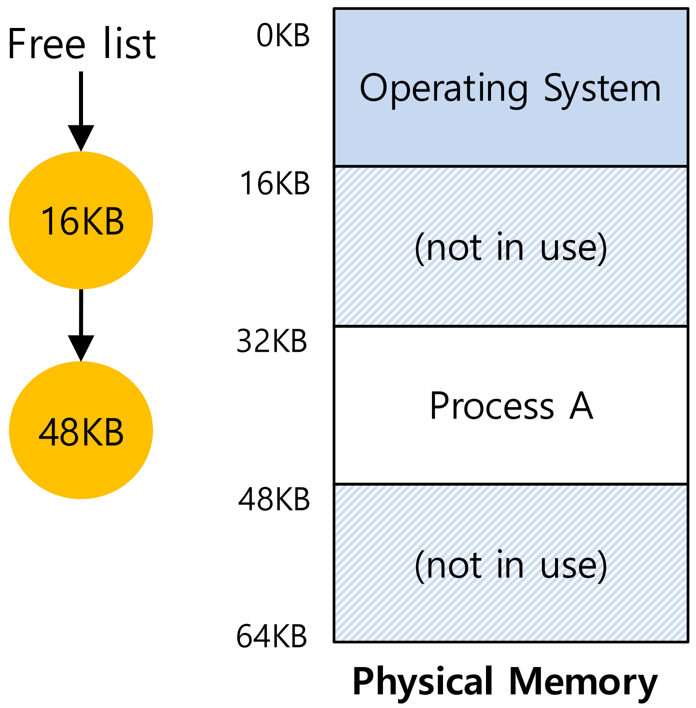
OS Issues: When a Process Starts Running

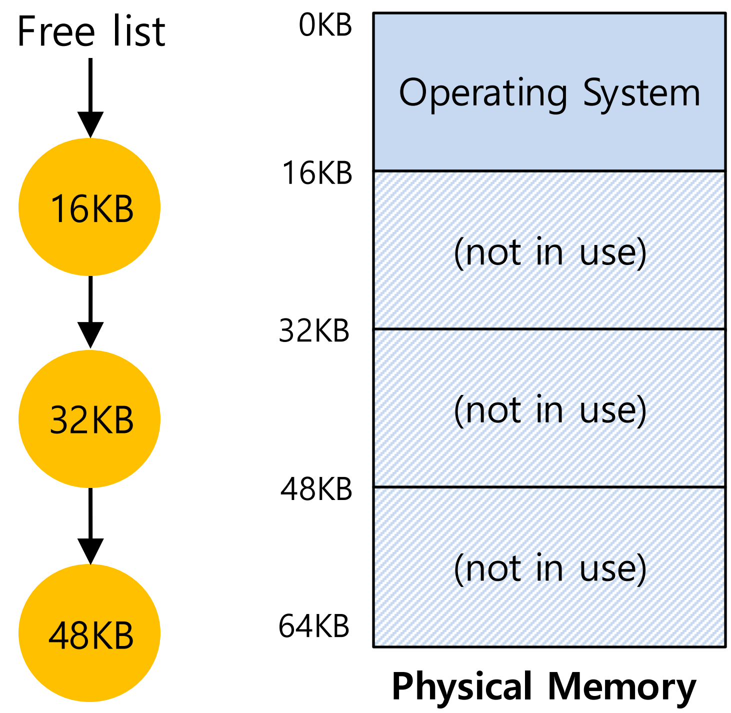
- The OS must **find a room** for a new address space
  - free list : A list of the ranges of physical memory which are not in use.



OS Issues: When a Process Is Terminated

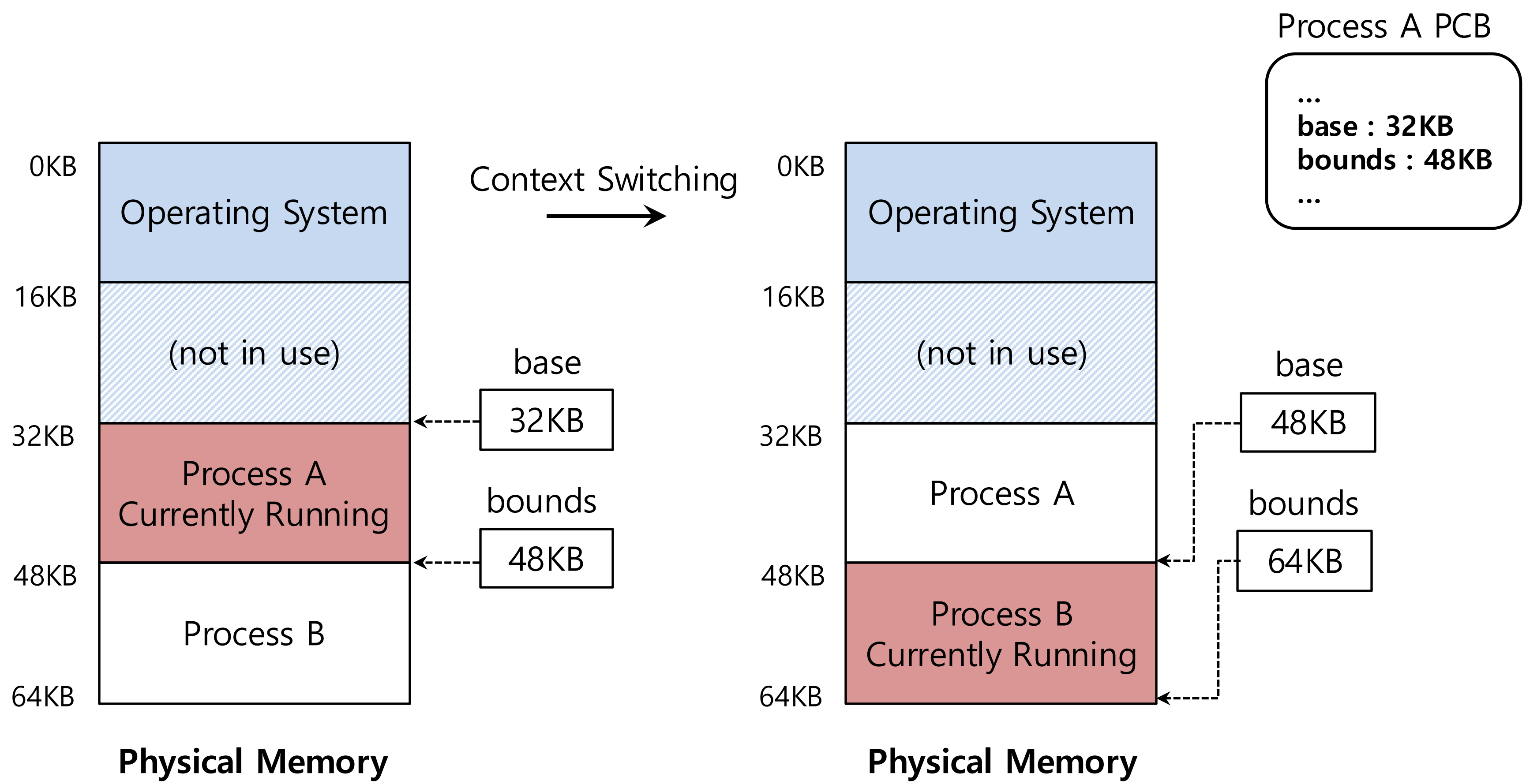
- The OS must **put the memory back** on the free list.





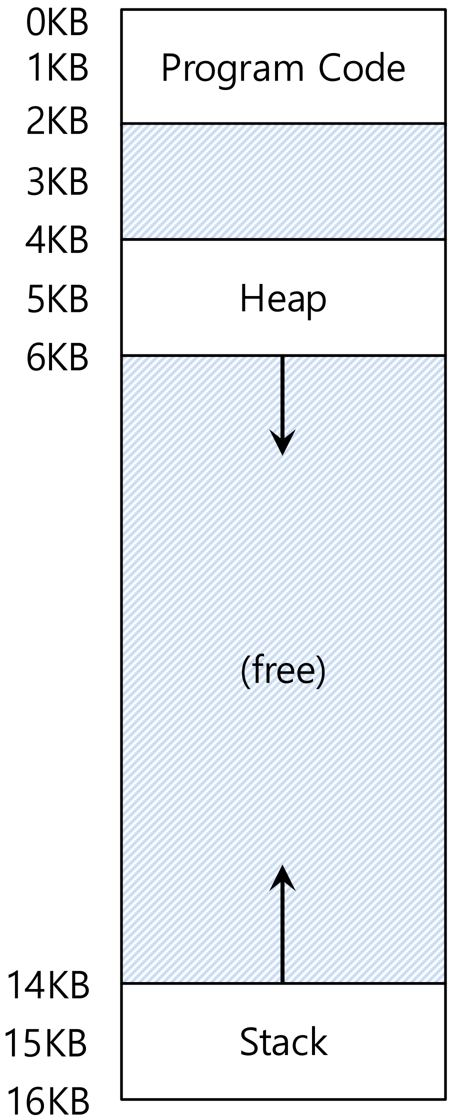
OS Issues: When Context Switch Occurs

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



Segmentation

Inefficiency of Base-and-Bounds

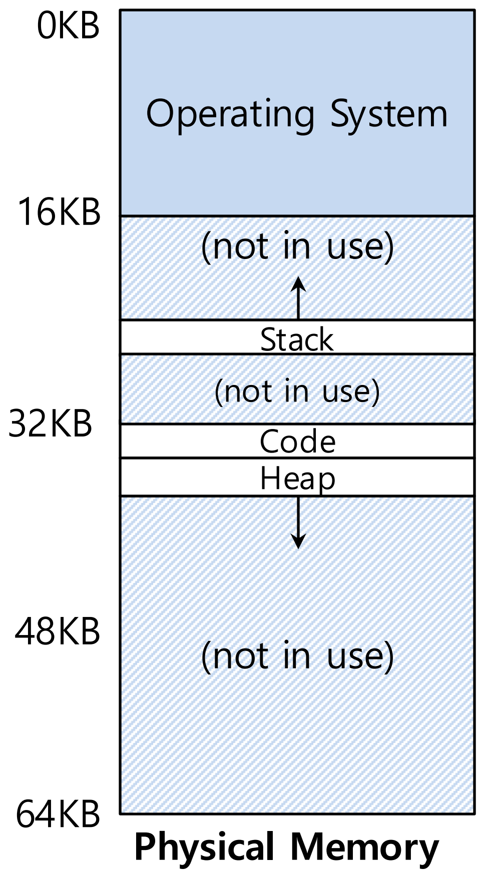


- **Big chunk of "free"** space
- "free" space **takes up** physical memory
- Hard to run when an address space **does not fit** into physical memory

Segmentation

- A segment is just **a contiguous portion** of the address space of a particular length
  - Logically different segments: code, stack, heap
- Each segment can be **placed** in **different parts of the physical memory**
  - **Base** and **bounds** exist for **each** segment

Placing Segments in Physical Memory



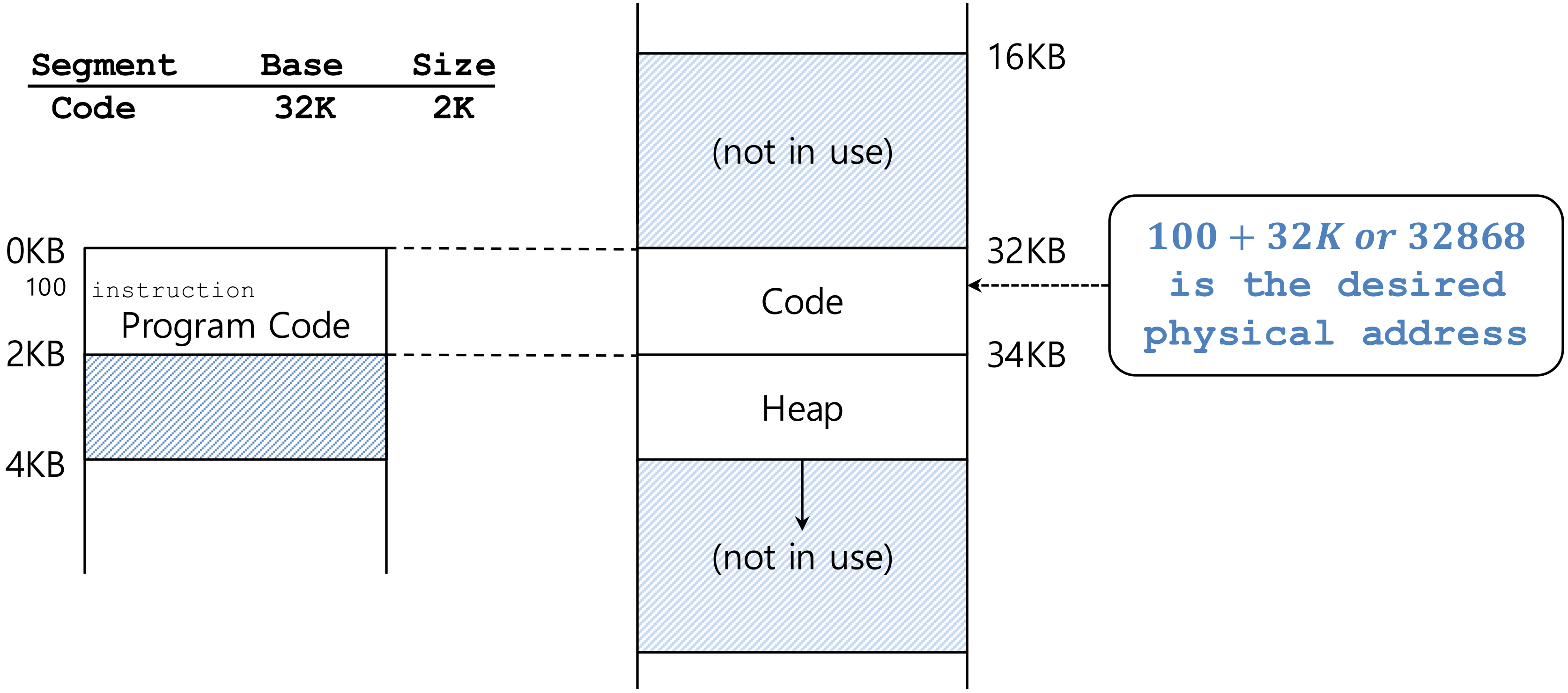
Segment	Base	Size
Code	32K	2K
Heap	34K	2K
Stack	28K	2K

Address Translation with 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

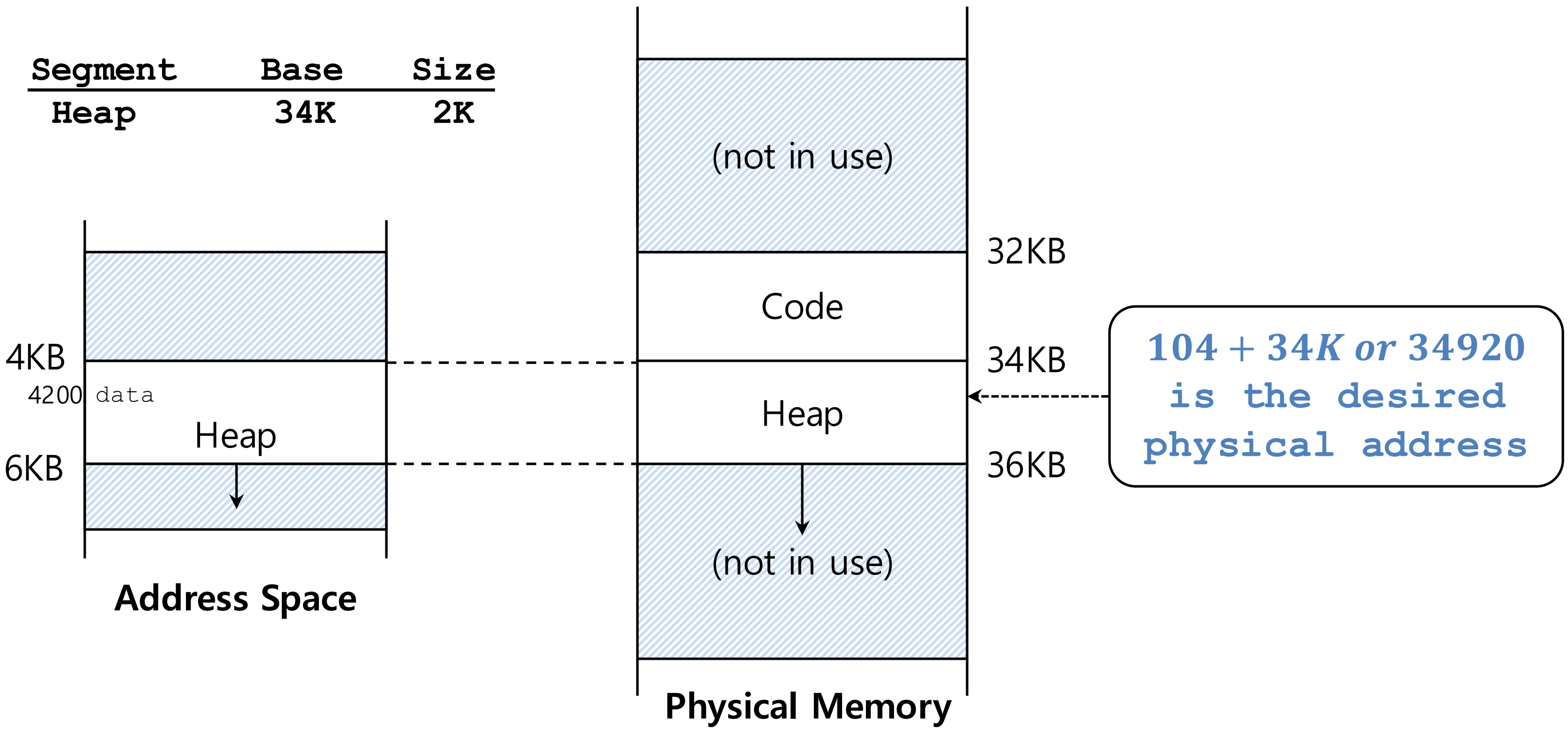


Address Translation on Segmentation: heap

$Virtual\ address + base$ is not the correct physical address.
$(Offset\ of\ virtual\ address) + base$ is the correct physical address.

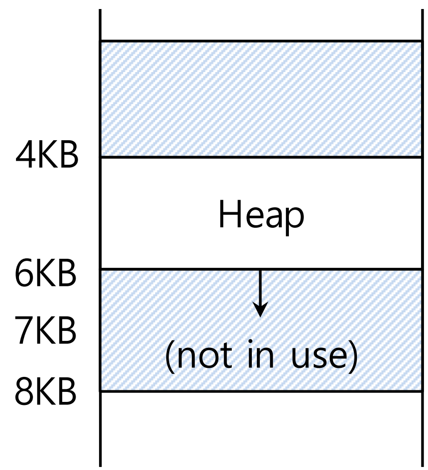
- The `offset` of virtual address 4200 is `104 (4200-4096)`
  - The heap segment **starts at virtual address 4096** in address space.

Segment	Base	Size
Heap	34K	2K



Segmentation Fault or Violation

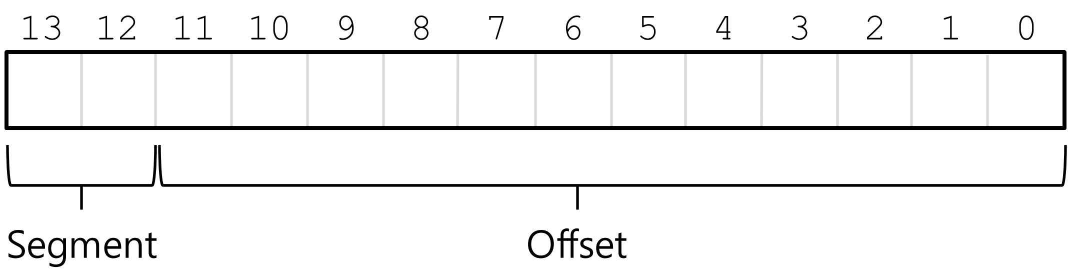
- If an **illegal address** such as 7KB which is beyond the end of heap is referenced, the OS occurs **segmentation fault**.
  - The hardware detects that address is **out of bounds**.



Address Space

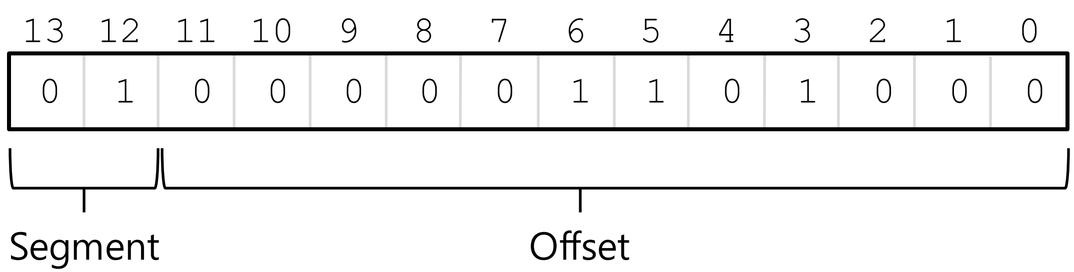
Referring to A Segment

- **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



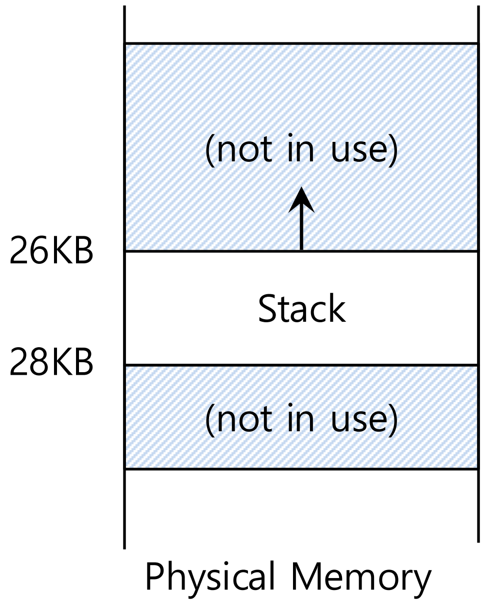
Segment Selection

```
// 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 the Stack Segment

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



Segment Registers (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 spaces
  - **Code sharing** is still in use in systems today
- 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)

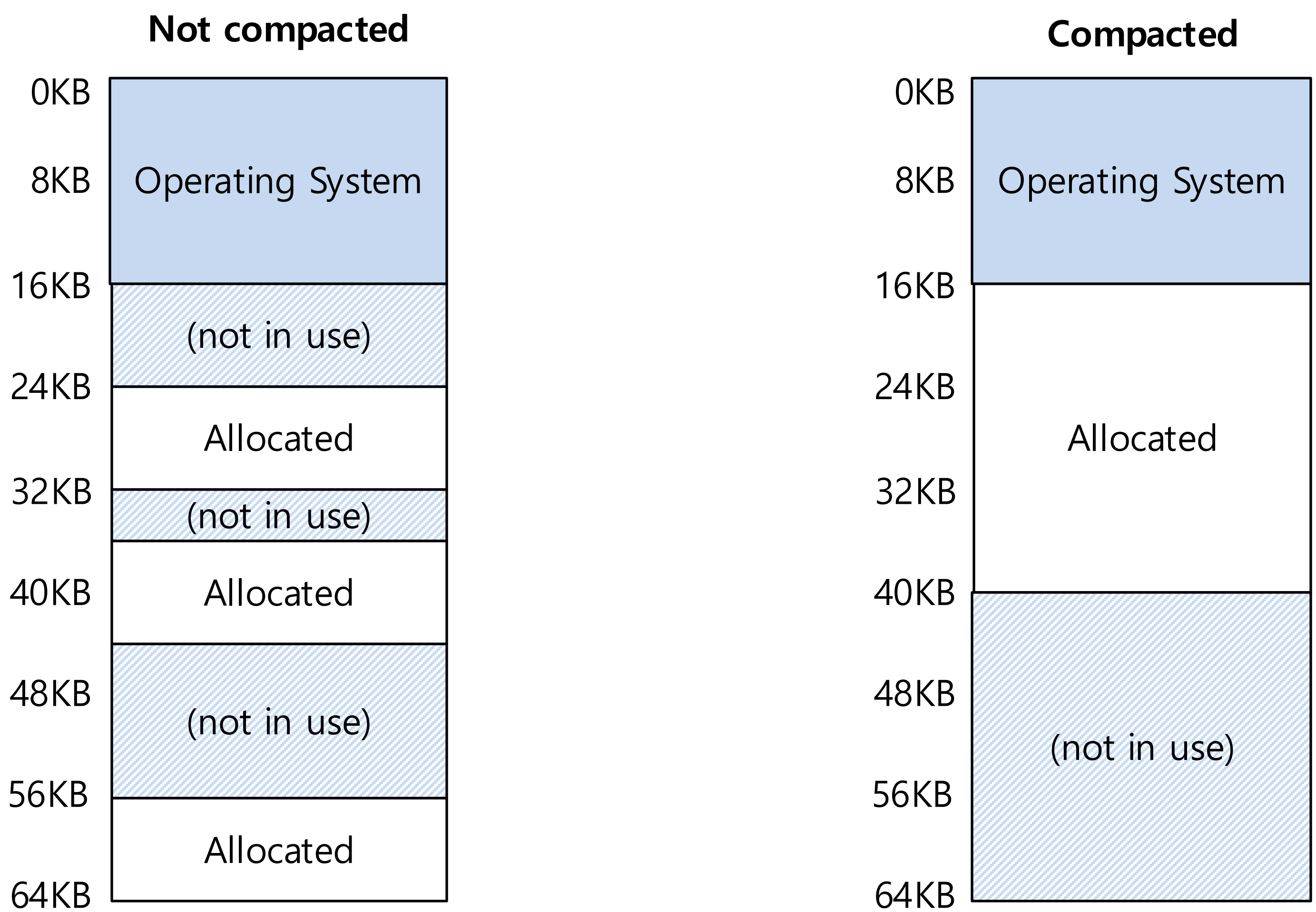
Segment	Base	Size	Grows Positive?	Protection
Code	32K	2K	1	Read-Execute
Heap	34K	2K	1	Read-Write
Stack	28K	2K	0	Read-Write

OS Support: Fragmentation

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

- **Compaction**: **re-arranging** the exiting segments
  - Compaction is **costly**
    - **Stop** running processes
    - **Copy** data to somewhere
    - **Change** segment register values

Memory Compaction



Issues of Segmentation

- External fragmentation
- Still inefficient and inflexible, when a segment is only sparsely used
- Better solution? Yes!