CDA0017: Operating Systems

Donghyun Kang (donghyun@changwon.ac.kr)

NOSLab (https://noslab.github.io)

Changwon National University

메모리 가상화

- 메모리 가상화란?
  - 물리 메모리를 가상화 하는 것
  - 각 프로세스가 전체 메모리를 사용하고 있다고 생각하게 하는 것

가상 메모리 설계의 목표

- 투명성(transparency)
  - 가상 메모리의 존재를 인지하지 못하도록 설계
- 효율성(efficiency)
  - 시간과 공간측면에서 효율적으로 설계
    - TLB 등의 하드웨어 지원이 필요함
- 보호(protection)
  - 프로세스를 다른 프로세스로부터 보호하도록 설계
    - **고립(isolate)**: 프로세스가 탑재, 저장, 혹은 명령어 반입 등을 실행할 때 다른 프로세스나 운영체제의 메모리 내용에 접근이 불가능 해야함

초기 메모리 시스템

- 메모리 가상화를 지원하지 않는 메모리 시스템
  - 효율성 & 사용성이 낮음

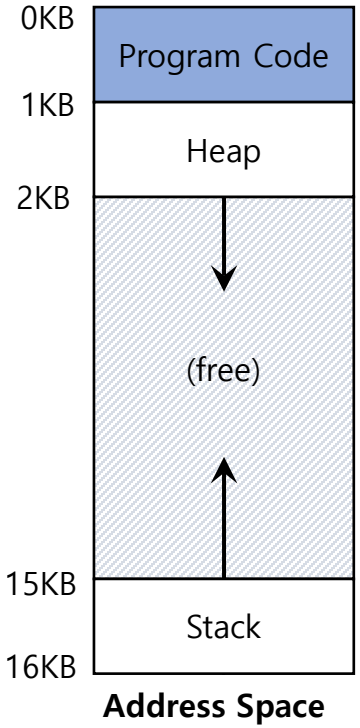
멀티 프로그래밍과 시분할 프로그래밍의 메모리 시스템

- 프로세스 전환 시 프로세스를 메모리에 그대로 유지
  - 짧은 시간 하나의 프로세스 실행
  - 메모리에 적재된 프로세스의 위치 교환
  - 효율성 & 사용성이 높음

- 그런, 프로세스 메모리 보호 문제 발생

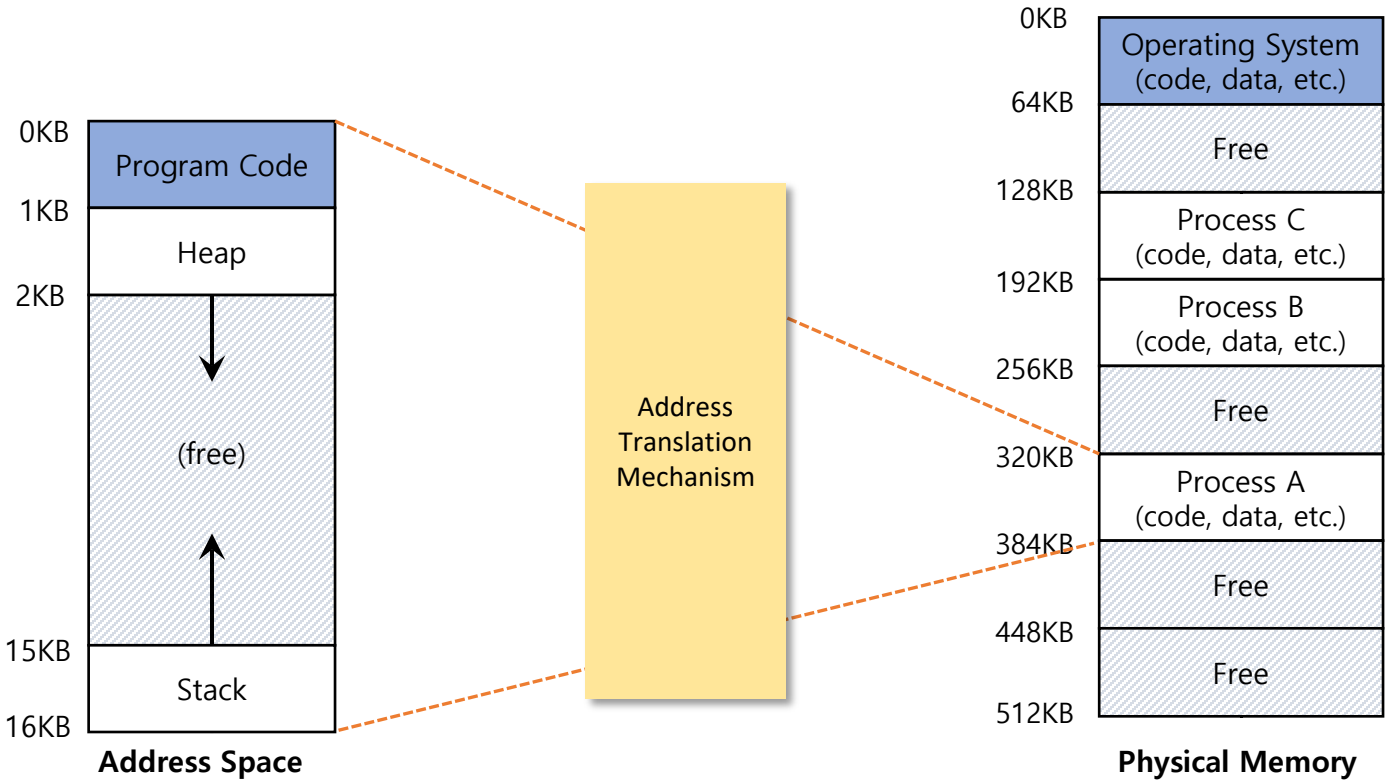
주소 공간

- 운영체제는 물리메모리의 추상화(주소공간)를 제공함
- 주소 공간 (address space)은 실행 프로그램의 모든 메모리 정보를 나타냄
  - 코드(code, 명령어)
    - 프로세스 실행 시 메모리 적재
  - 스택 (stack)
    - 현재 위치, 지역 변수, 함수 인자와 반환 값
  - 힙 (heap)
    - 동적으로 할당되는 메모리



메모리를 어떻게 가상화하는가?

- 가상 주소를 물리 주소로 변환 (address translation)



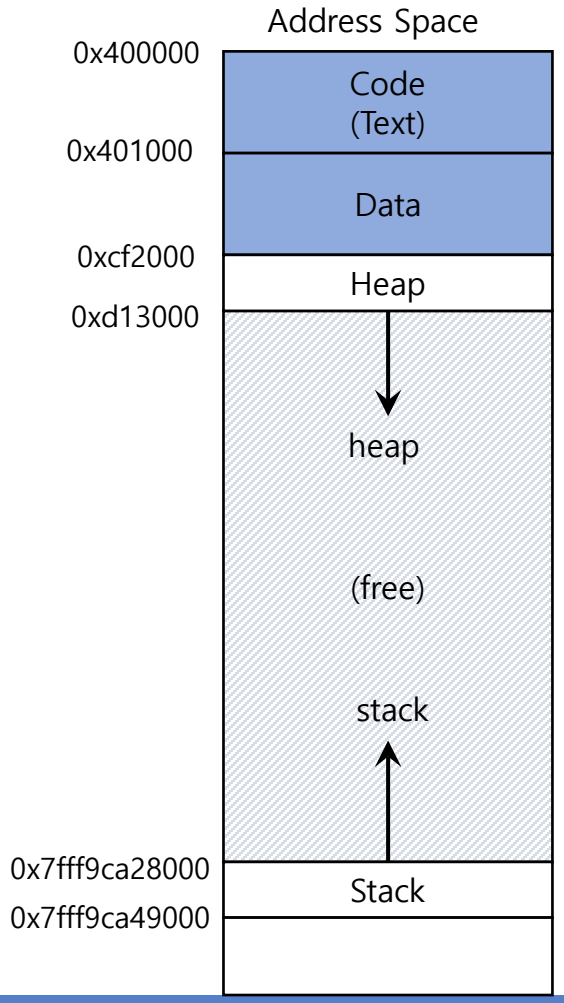
가상 주소

- 프로그램의 모든 주소는 가상 주소 공간임
  - 운영체제는 가상 주소를 물리 주소로 변환함

```
#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;
}
```

가상 주소

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



주소 변환

- 가상 주소는 하드웨어를 통해 물리 주소로 변환됨
  - 실제 프로세스가 적재된 물리 메모리의 주소를 저장해야함

- 가정
  1. 사용자 주소 공간은 물리 메모리에 연속적으로 배치되어야 함
  2. 주소 공간의 크기가 너무 크지 않으며, 물리 메모리 크기보다 작음
  3. 각 프로그램의 주소 공간의 크기는 같음 (16 KB)

사례

- C 언어 기반 프로그램 작성

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

  - 메모리 접근 (load)
  - 더하기 연산
  - 메모리 접근 (store)

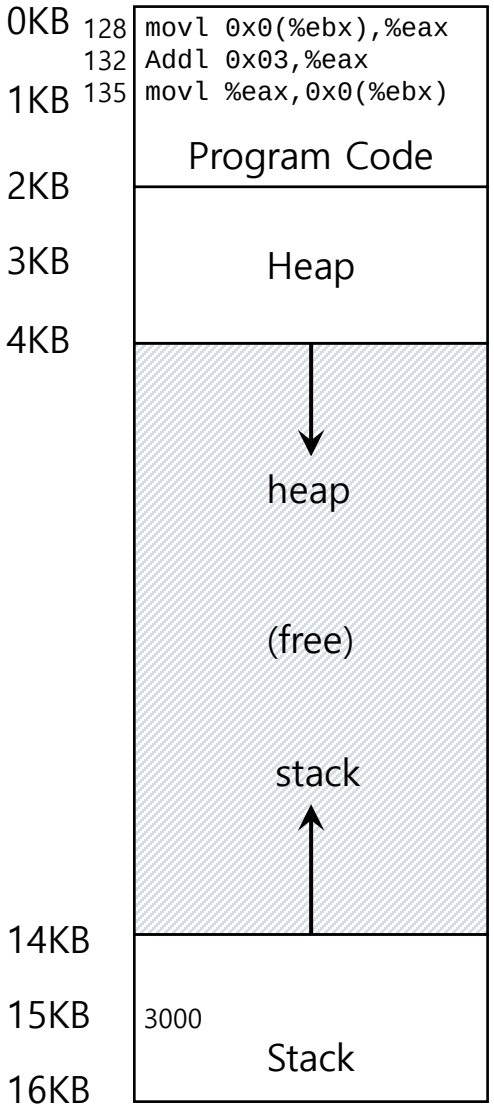
사례

- 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
```

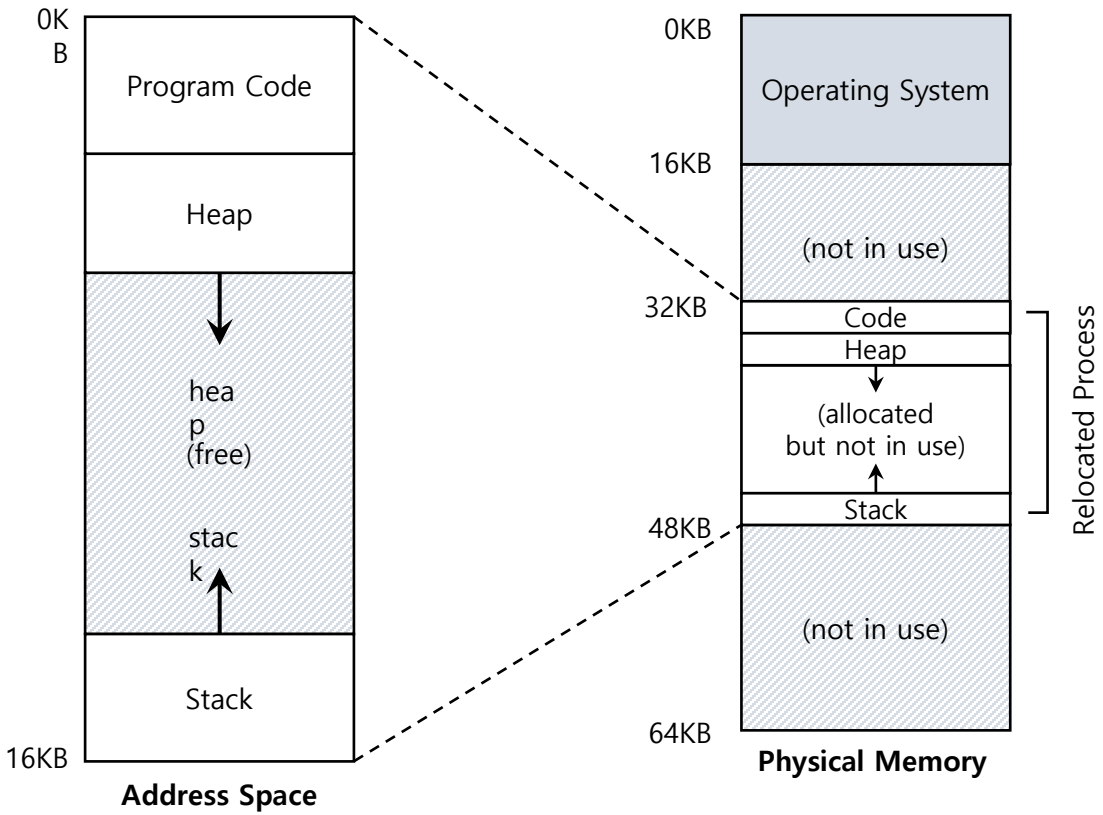
사례

- 실행 과정
  - 주소 128의 명령어를 반입
  - 이 명령어 실행 (주소 15 KB에서 탑재 ← **x 가 저장된 위치**)
  - 주소 132의 명령어를 반입
  - 이 명령어 실행 (메모리 참조 없음)
  - 주소 135의 명령어를 반입
  - 이 명령어 실행 (15 KB에 저장 ← **x가 저장된 위치**)



주소 공간 재배치

- 운영체제는 물리메모리 번지에 프로세스를 적재함
  - 그러나, 물리 메모리가 0 번지 주소가 아님



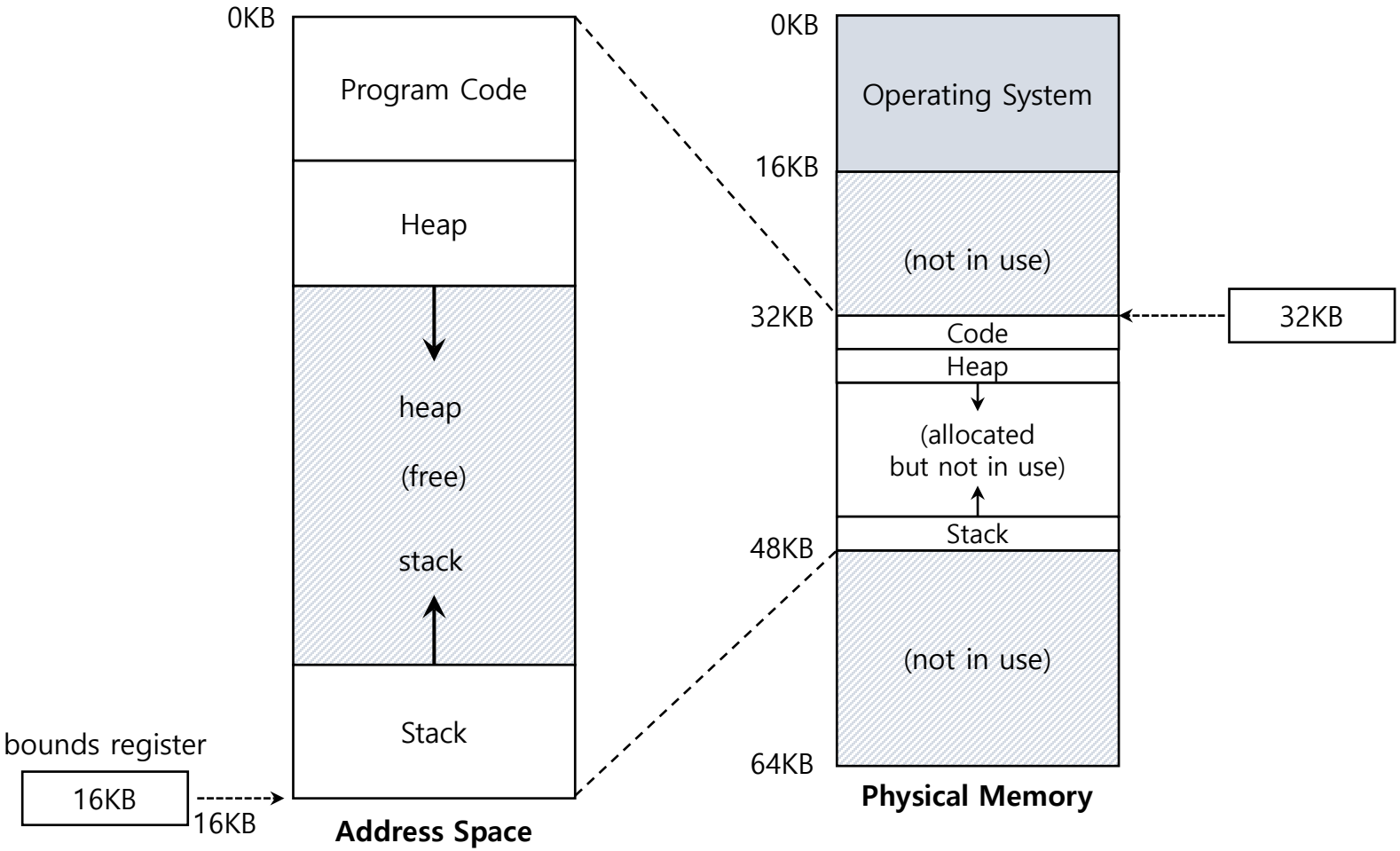
동적 (하드웨어-기반) 재배치

- MMU (Memory Management Unit) performs address translation on every memory reference instructions
- Protection is enforced by hardware: if the virtual address is invalid, the MMU raises an exception
- OS passes the information about the valid address space of the current process to the MMU

동적 (하드웨어-기반) 재배치

- MMU (Memory Management Unit) performs address translation on every memory reference instructions
- Protection is enforced by hardware: if the virtual address is invalid, the MMU raises an exception
- OS passes the information about the valid address space of the current process to the MMU

베이스와 바운드(base and bound)



동적 (하드웨어-기반) 재배치

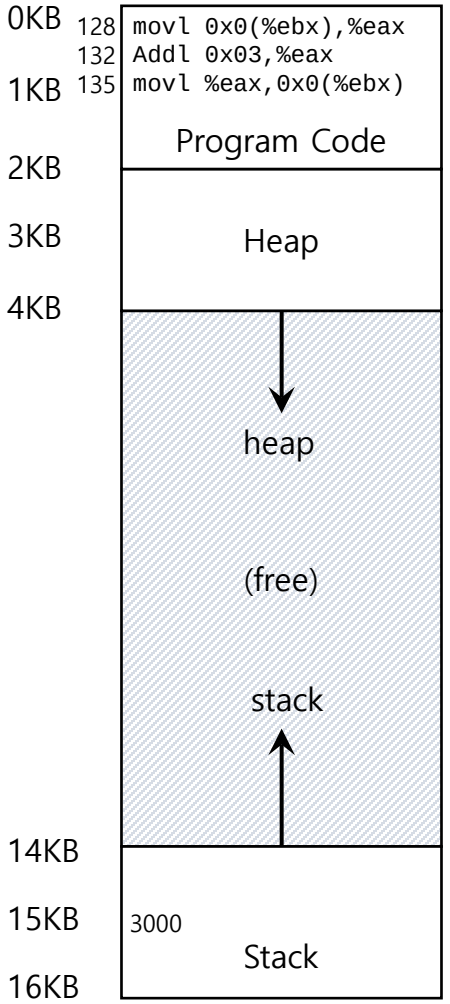
- 각 프로그램은 주소 0에 탑재되는 것처럼 작성되고 컴파일됨
- 프로세스가 실행되면 프로그램의 모든 주소가 프로세서에 의해 변환됨
  - 물리 주소 (physical address) = 가상 주소 (virtual address) + 베이스 (base)
- 메모리 주소의 잘못된 접근
  - 0 <= 가상 메모리 < 바운드(bound)

사례

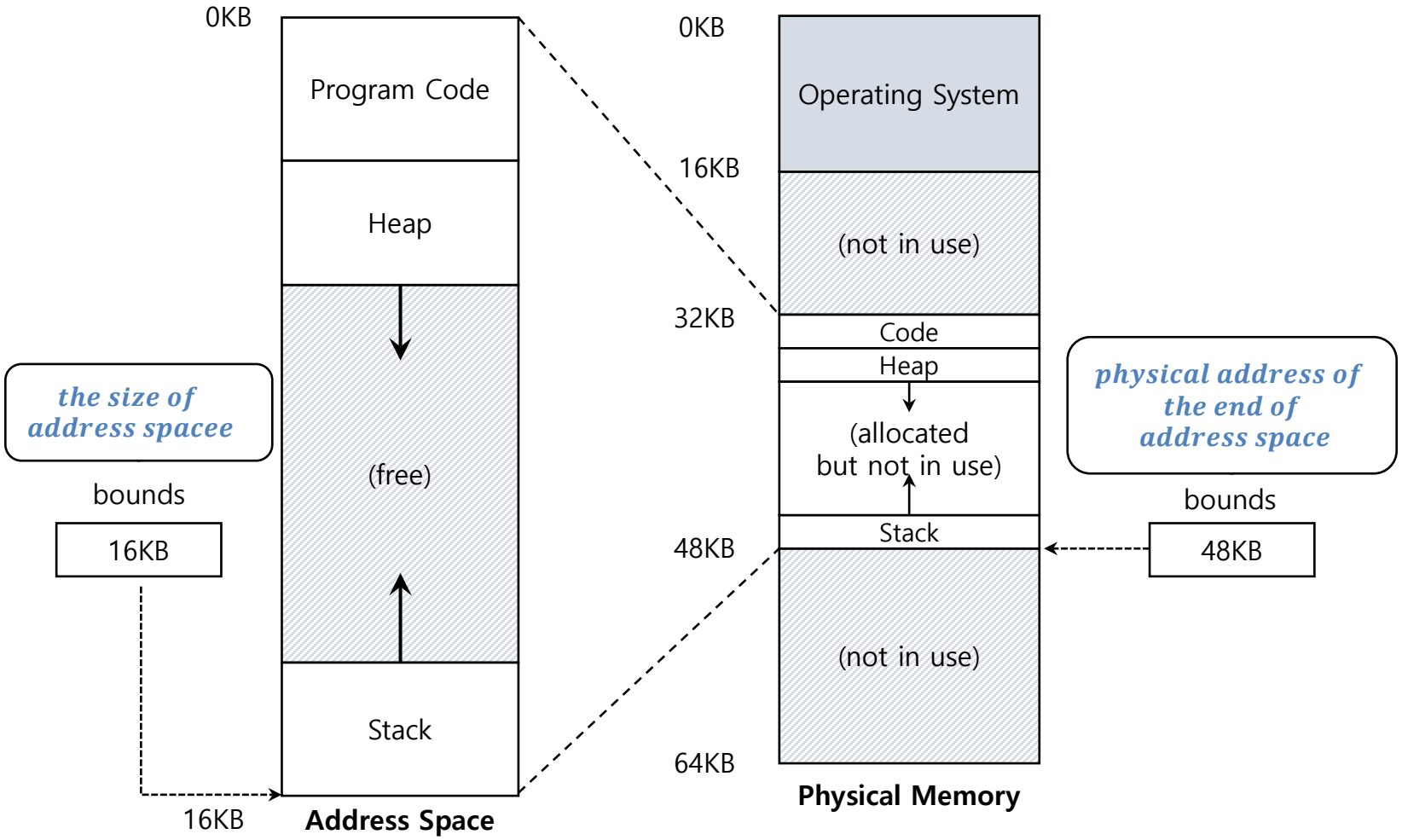
```
128 : movl 0x0(%ebx), %eax
```

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

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



바운드 레지스터의 관리

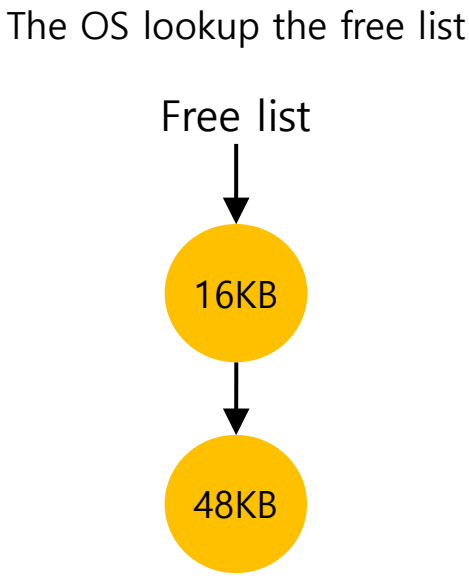

0KB
Program Code
Heap
(free)
Stack
16KB
Address Space
the size of address spacee
bounds
16KB
0KB
Operating System
16KB
(not in use)
32KB
Code
Heap
(allocated but not in use)
Stack
48KB
(not in use)
64KB
Physical Memory
physical address of the end of address space
bounds
48KB

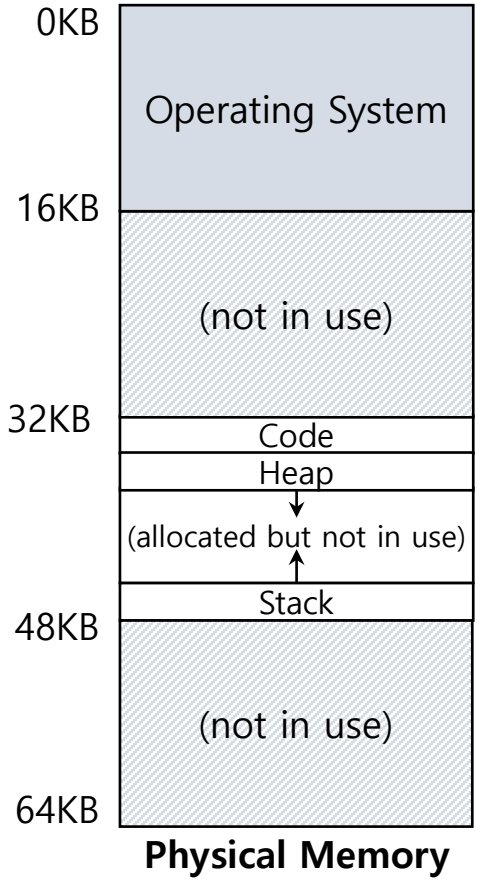
운영체제 이슈

- 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

프로세스 시작

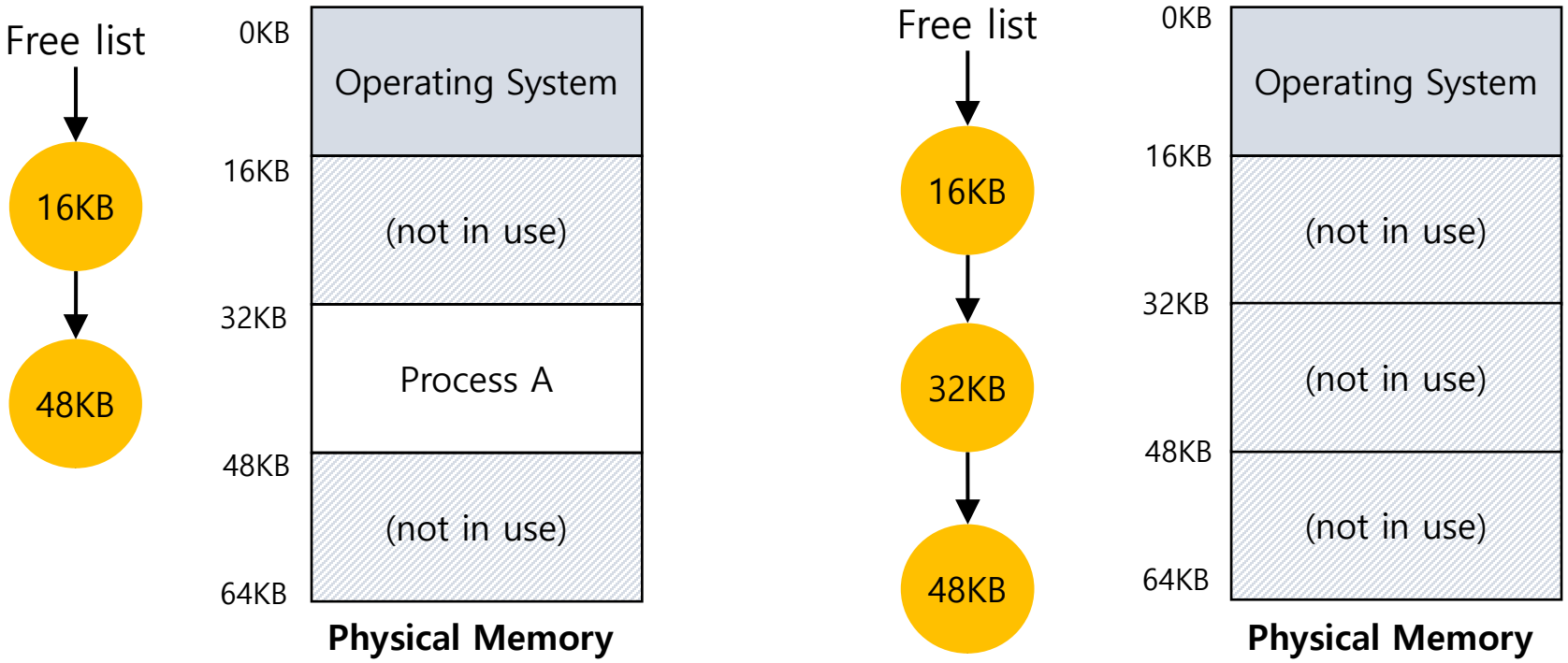
- 운영체제는 빈 공간을 찾아야 함





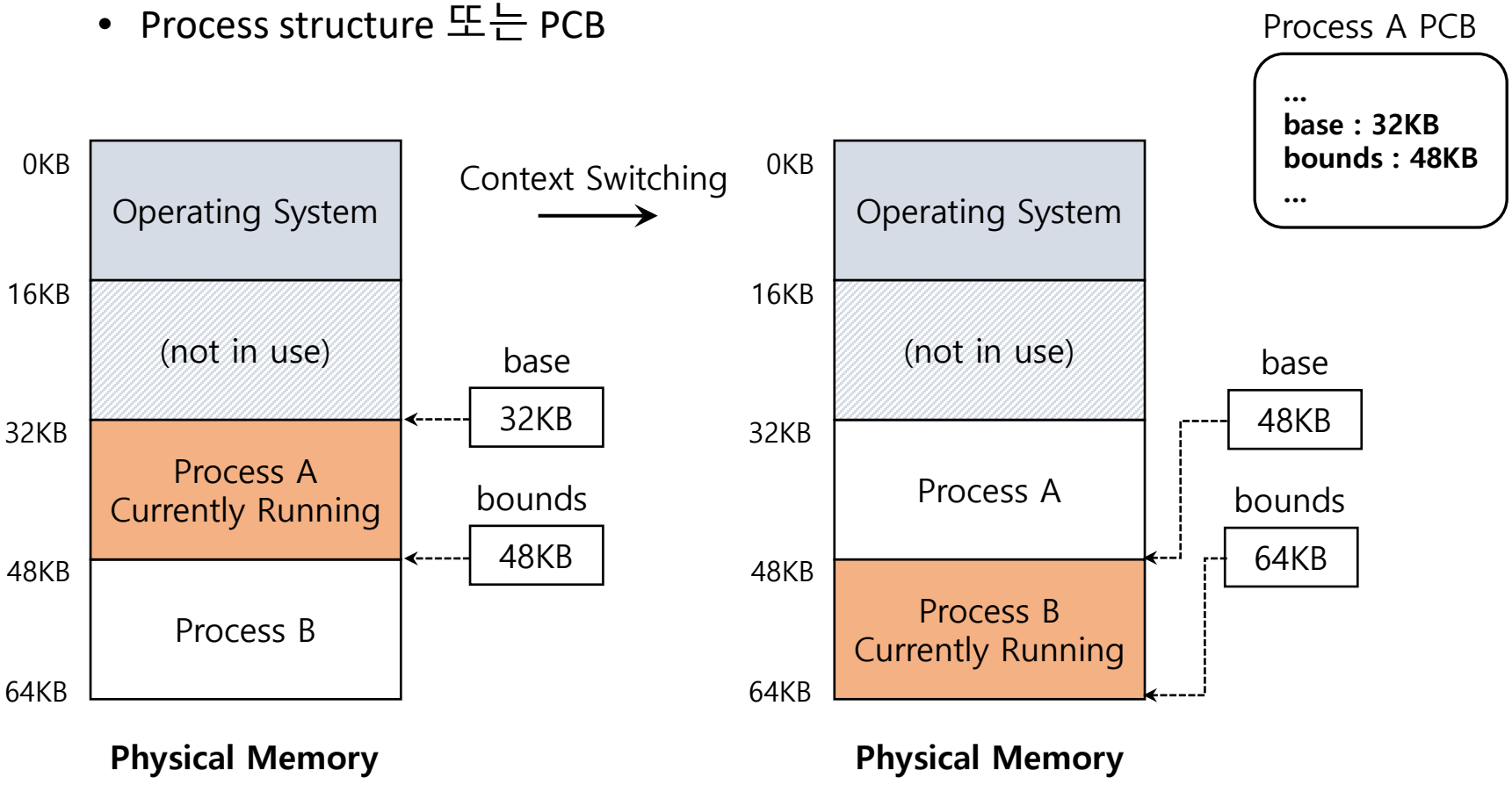
프로세스 종료

- 운영체제는 사용된 공간을 반환 해야함



프로세스 교환 (context switch)

- 운영체제는 베이스와 바운드 레지스터를 저장/복원 해야함
  - Process structure 또는 PCB



세그먼테이션

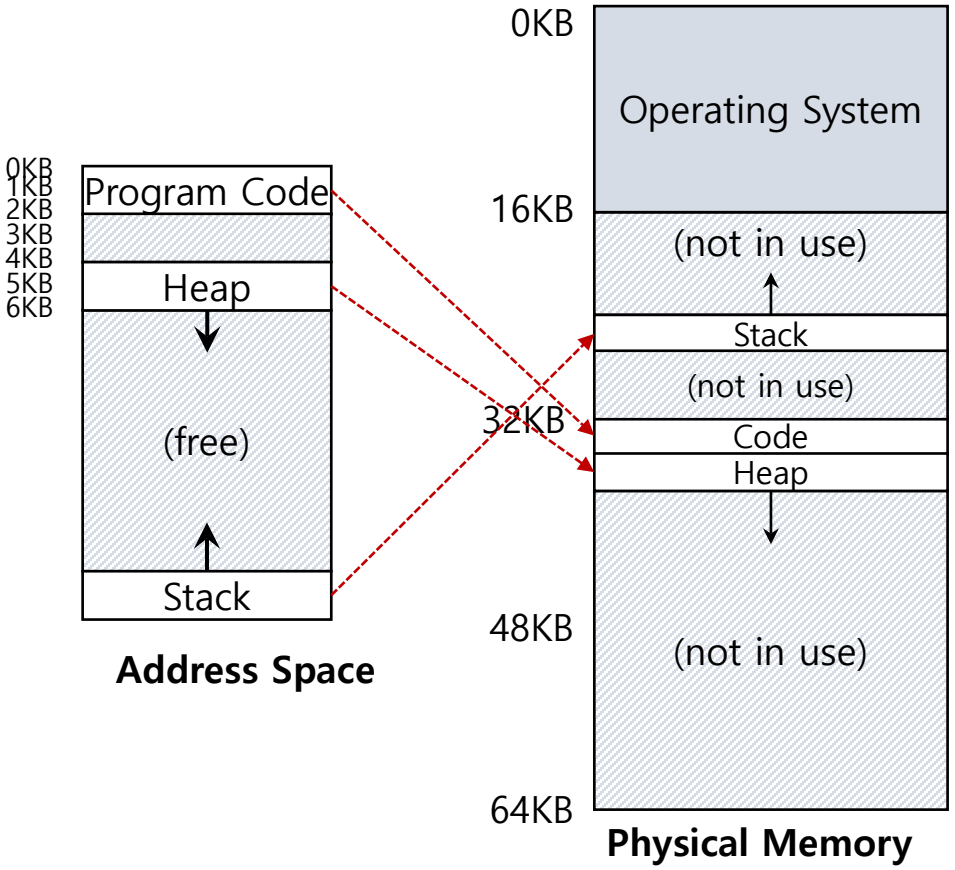
메모리는 효율적으로 사용되고 있는가?

- 스택과 힙 영역 사이에 잠재적으로 빈 영역 존재
- 수 바이트를 위해 수십 KB을 낭비함
- 프로그램 주소 공간이 물리 메모리보다 큰 경우 실행 불가

세그멘테이션(segmentation)

- 기존 주소 체계보다 세분화된 세그먼트 (segment) 도입
  - 1960년대 아이디어 도입
  - 특정 길이를 가진 연속적인 주소 공간
    - 세그먼트 베이스
    - 세그먼트 바운드
  - 세그먼트 단위 메모리 배치
    - 세그먼트 종류: 코드, 스택, 힙
    - 각 세그먼트를 물리 메모리의 다른 위치에 배치

세그멘테이션(segmentation) 예제



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

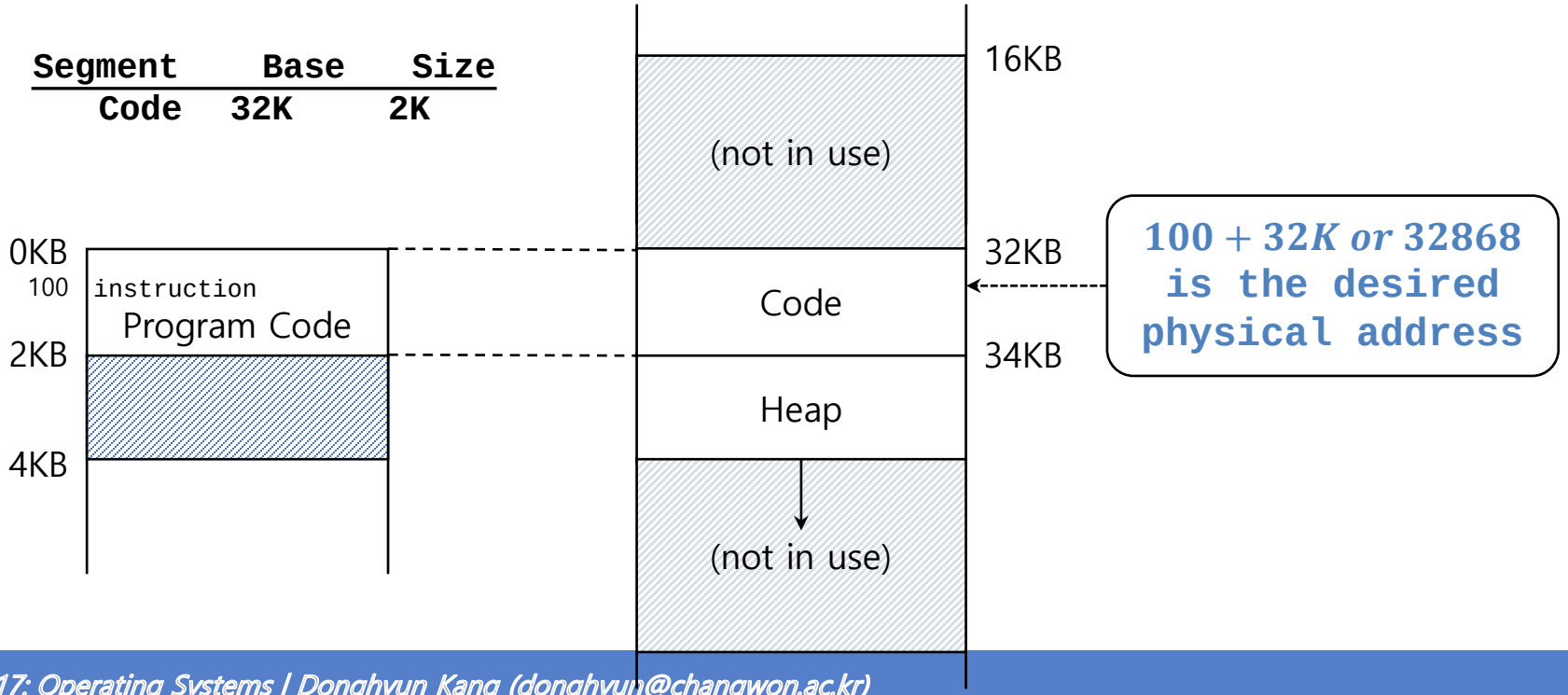
$$phycal\ address = offset + base$$

$$offset = virtual\ address - start\ address\ of\ segment$$

세그멘테이션(segmentation) 예제

$$physical\ address = offset + base$$

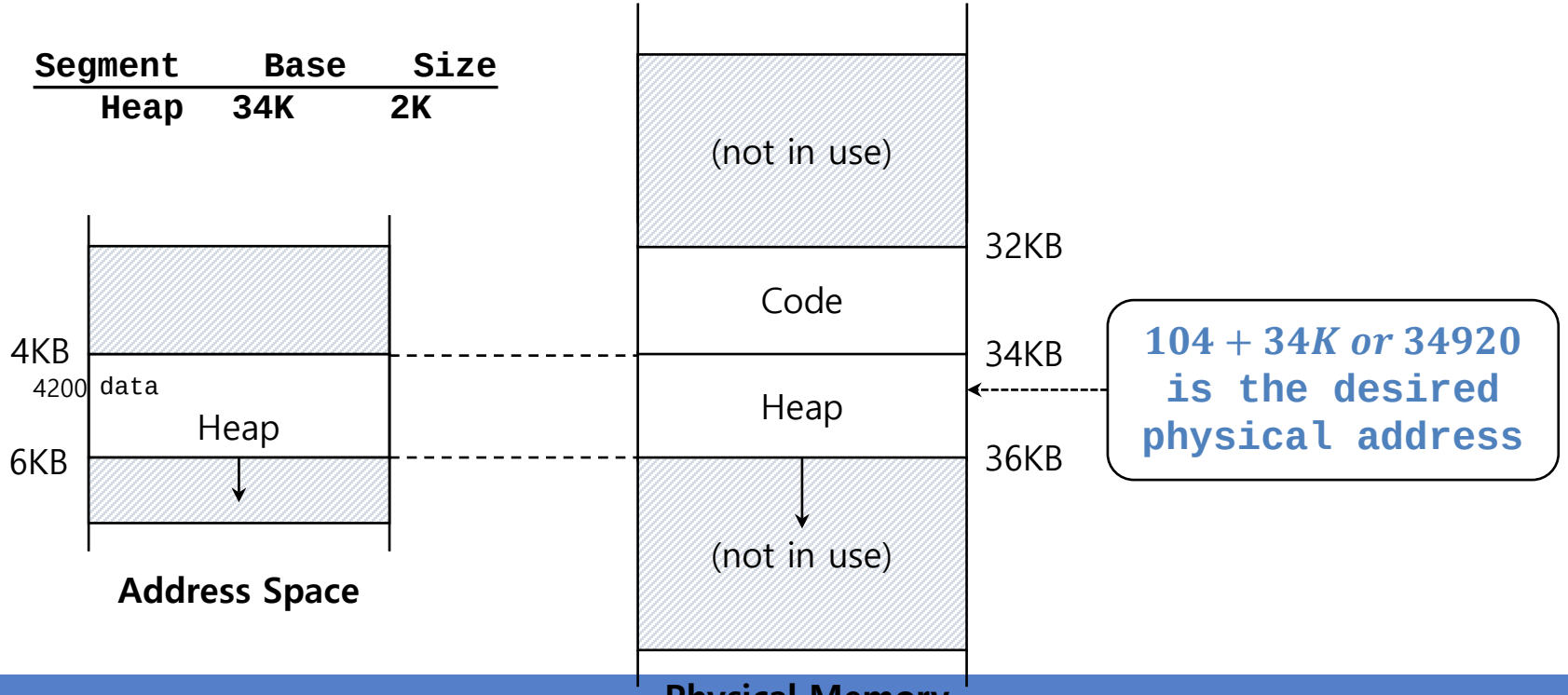
- The `offset` of virtual address 100 is `100`.
  - The code segment **starts at virtual address 0** in address space.



세그멘테이션(segmentation) 예제

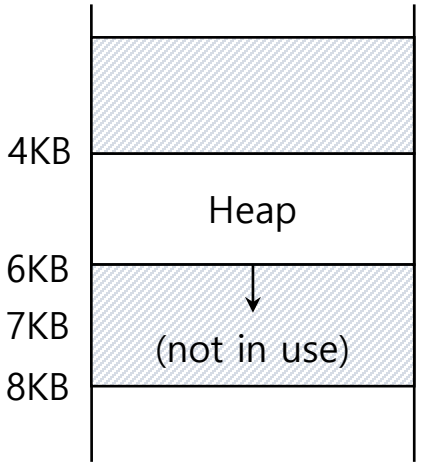
Virtual address + *base* is not the correct physical address.

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



세그먼트 폴트

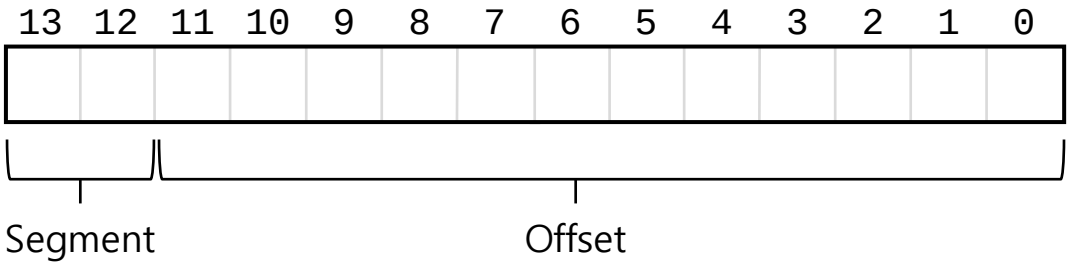
- If an illegal address such as 7KB which is beyond the end of heap is referenced, the OS occurs segmentation fault.
  - The hardware detects that address is **out of bounds**.



Address Space

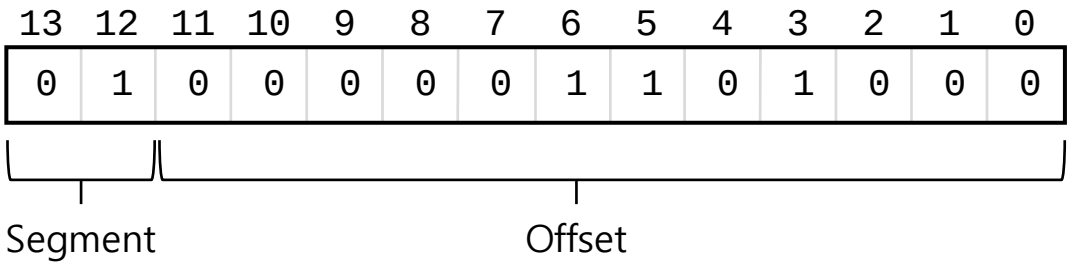
세그먼트 참조

- 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
-	10
Stack	11



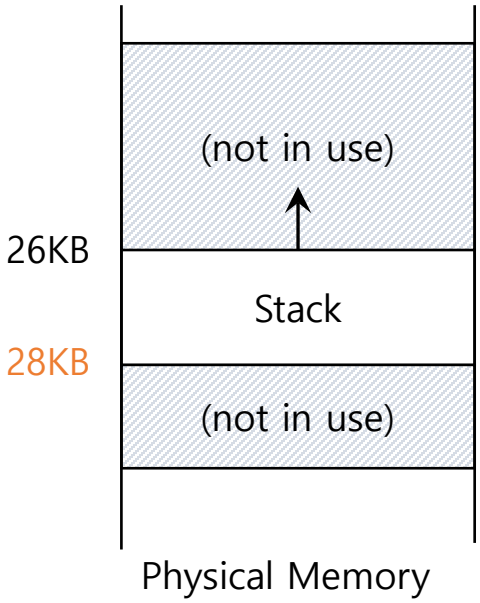
Referring to Segment(Cont.)

```
1   // get top 2 bits of 14-bit VA
2   Segment = (VirtualAddress & SEG_MASK) >> SEG_SHIFT
3   // now get offset
4   Offset = VirtualAddress & OFFSET_MASK
5   if (Offset >= Bounds[Segment])
6        RaiseException(PROTECTION_FAULT)
7   else
8        PhysAddr = Base[Segment] + Offset
9        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



Segment Register(with Negative-Growth Support)

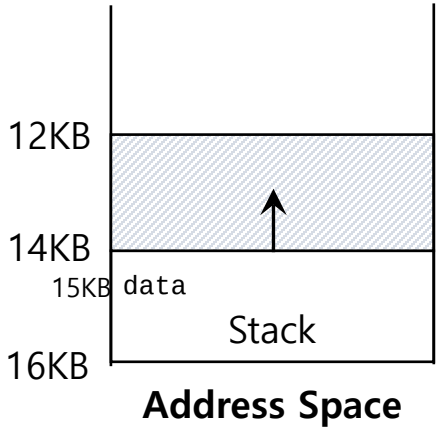
Segment	Base	Size	Grows Positive?
Code	32K	2K	1
Heap	34K	2K	1
Stack	28K	2K	0

Referring to Stack Segment (Cont'd)

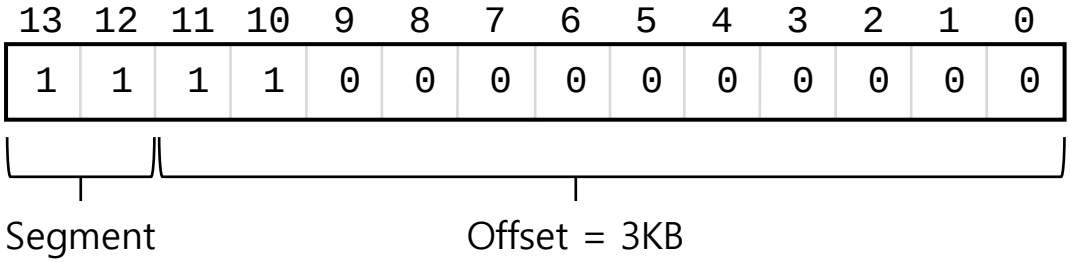
Segment Register(with Negative-Growth Support)

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

Segment	bits
Code	00
Heap	01
-	10
Stack	11



virtual address 15KB = 11 1100 0000 0000



Max. segment size = 4KB (because of 12bit offset)

Correct negative offset = 3KB – 4KB = -1KB

Physical address = 28KB – 1KB = 27KB

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

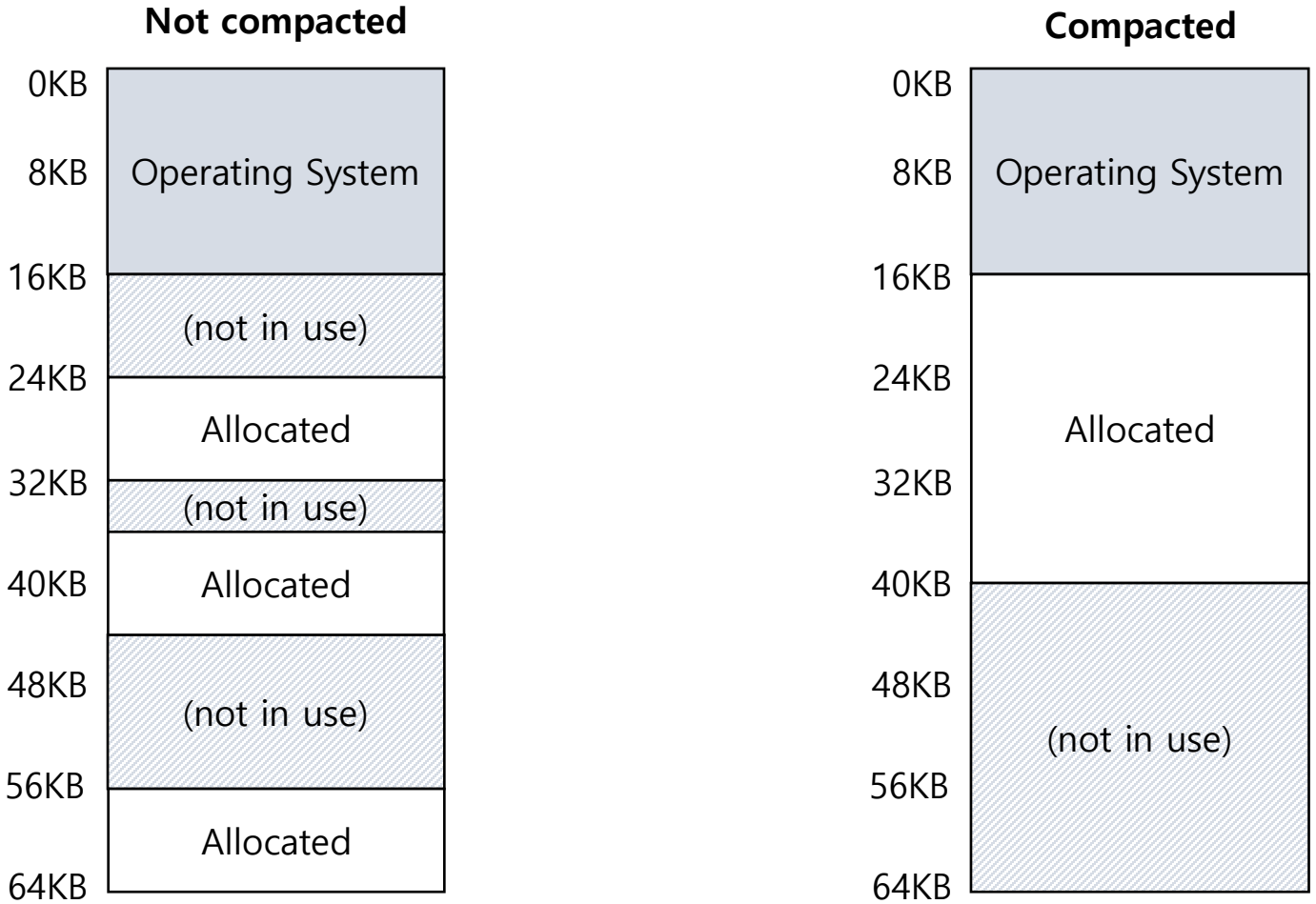
- Coarse-Grained means segmentation in a small number.
  - e.g., code, heap, stack.
- Fine-Grained segmentation allows more flexibility for address space in some early system.
  - To support many segments, Hardware support with a **segment table** is required.

OS support: Fragmentation

- External Fragmentation: little holes of free space in physical memory that make difficulty to allocate new segments.
  - 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



Q&A