08. Segmentation

▼ Segmentation ⇒메모리 사용 효율성↑

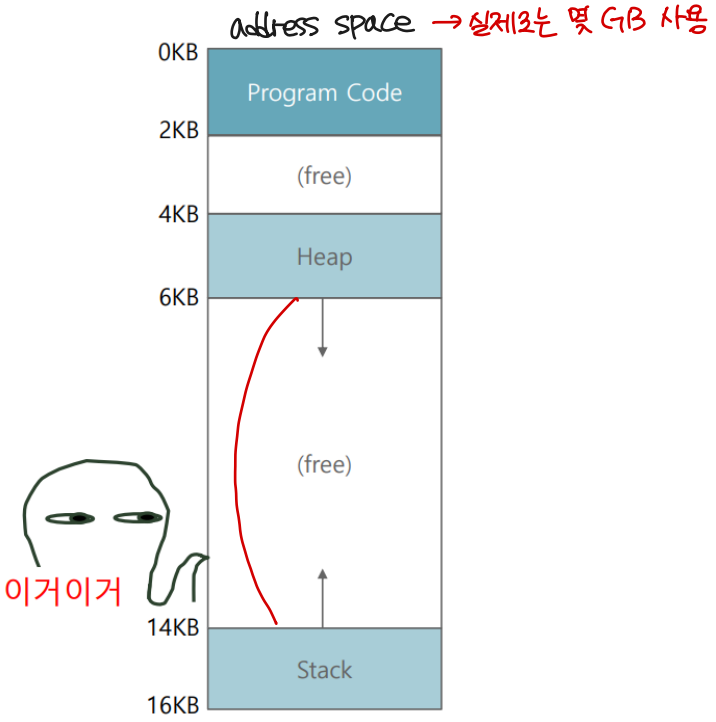
(segfault → segmentation rule을 무언가 어겼을 때의 rule)

⇒ 지금까지 address space가 physical memory의 연속적인 공간에 존재한다고 가정함

→ 이로 인해 생기는 문제를 segmentation으로 해결!

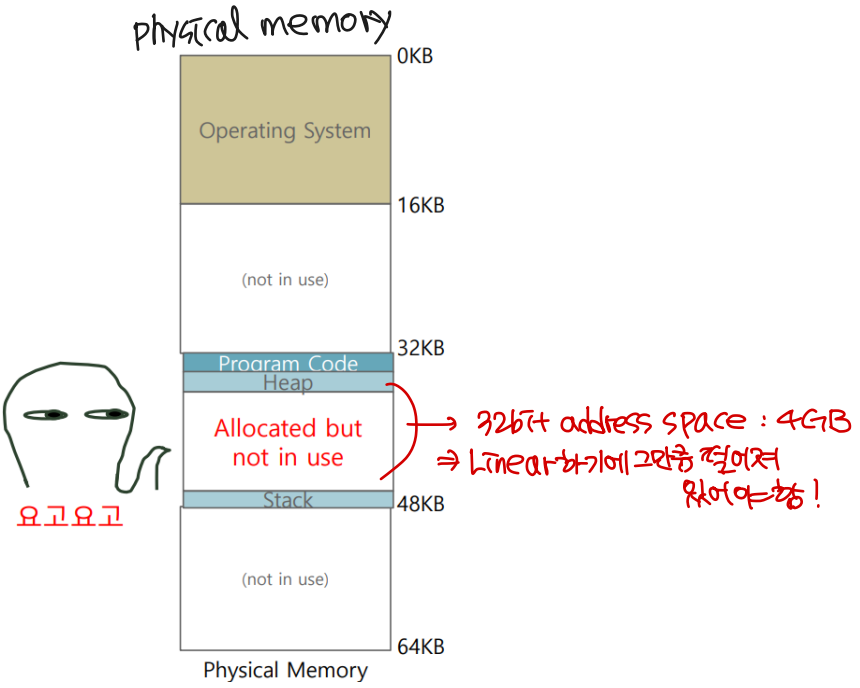
▼ Base and Bounding → 지금까지의 문제점

1. address space 클 때 ⇒ physical memory linear ⇒ 비효율적
2. Big chunk of 'free' space → 상당히 큰 영역이 사용되지 않고 있음



- 전체 address space를 physical memory의 어딘가로 재배치할 때 physical memory 차지
- 전체 address space가 memory에 딱 맞지 않을 때 program 돌리기 빡셈
- 실제로 사용하고 있는 process → address space 공간 크기가 큼 ⇒ 심각

⇒ 큰 address space 사용하고 싶을 때



⇒ 할당되지 않은 공간 때문에 physical memory가 부족한 경우가 발생할 수 있음

~~⇒ address space가 physical memory의 연속적인 공간에 존재한다는 가정~~

→ 현실적으로 어려움.

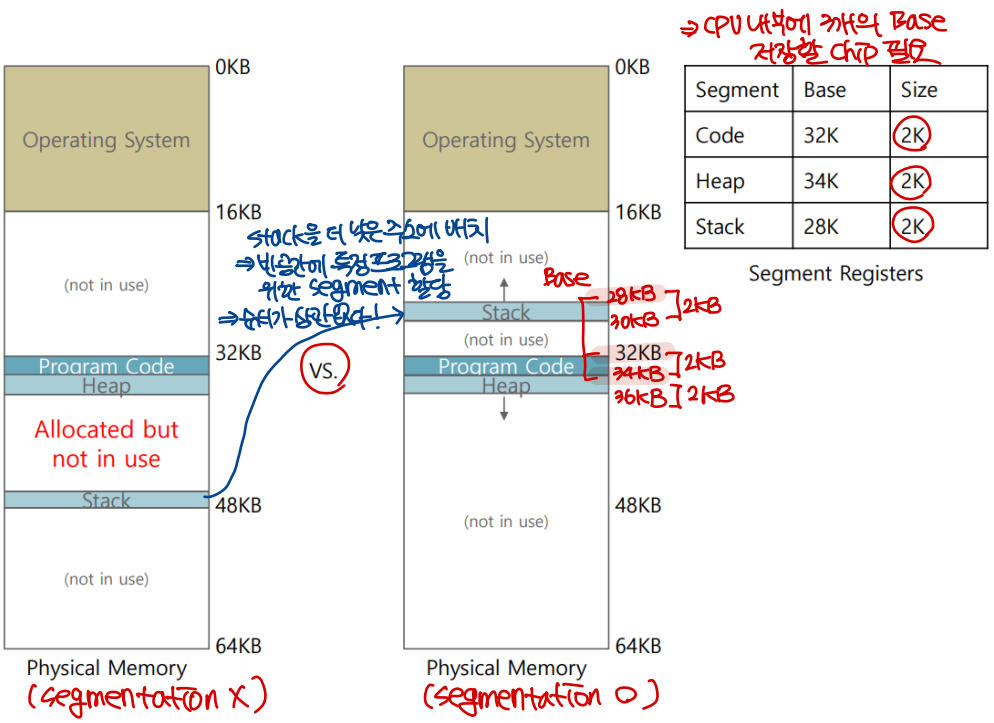
→ stack과 heap 사이의 많은 free space 활용하고 싶음!

▼segmentation → 이 방법으로 해결해보자!

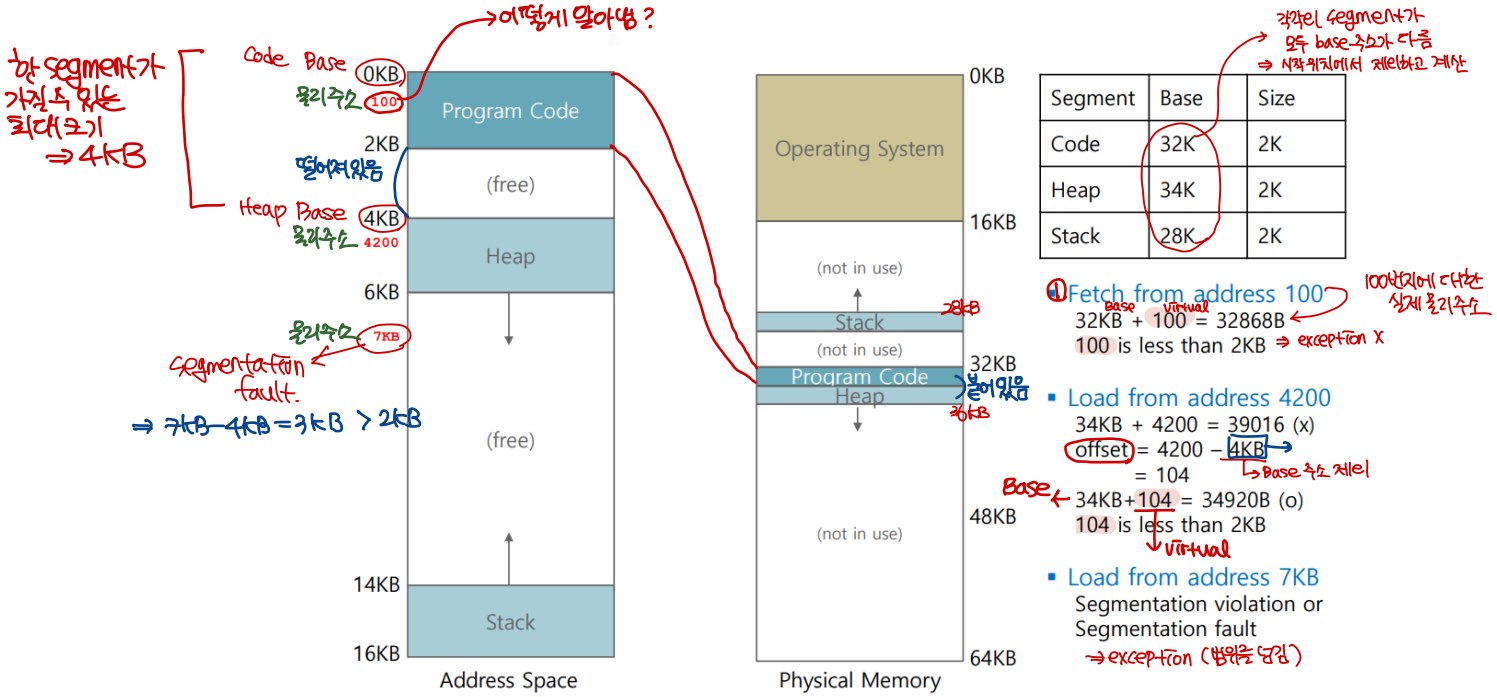
- segment
  - 조각 단위로만 linear하게 제공하는 것으로 해결 → segment 내부는 상관 x
  - 특정 길이의 address space의 연속된 부분이 생김 → (code, stack, heap)
  - 각 segement마다 base, bound 가지고 있음 ⇒ register가 많아짐

- **segmentation** → 각 segment마다 base, bound pair가 필요
  - physical memory에 다른 부분에 각 segment를 배치
  - code, heap 구분은 따로 하지 않음
  - ~~⇒~~ 사용하지 않는 virtual address space로 physical memory를 채우는 것을 방지
    - 오직 사용된 memory만 physical memory의 공간을 할당 받음
    - 사용하지 않는 large address space → 효율적으로 relocation 가능하다는 장점

- example → relocation



- example → address translation ⇒ hw가 진행하고 OS가 관리
  - 공간 할당 순서가 뒤죽박죽이어도 여전히 linear한 것처럼 실행됨



▼ segement별 주소 공간 표현법(address translation)

어떤 주소값에 대해서 어떤 segement를 기준으로 해야 하는지 어떻게 알 수 있을까?

- **explicit approach → code, heap segment 위주**

- virtual address의 상위 몇 개의 bit에 해당되는 segment address space를 조각내어서 넣어줌 → 정보 표시 !!

- example

  → predefined : 14bit addressing (16KB address space)

  $\Rightarrow 2^{14} = 2^4 \times 2^{10}B = 16KB$ (0,1)

13	12	11	10	9	8	7	6	5	4	3	2	1	0

Segment (bits 13–12) · Offset (bits 11–0)

0 or 1을 가질 수 있는 한 칸

  - 00 : code segment
  - 01 : heap segement
  - 11 : stack segement
  - 10 → unused (쓰면 문제 생길수도 있음)
    - 어떤 system은 한 bit만 사용함 → code, heap = 1 / stack = 0
    - heap, code가 같은 bit로 표현할 경우 → 둘의 공간이 서로 붙어 있어야

  ex) address 100 ⇒ 00 | 000001100100

  - max size of each segment : 4KB (각 segment 크기)
    - offset :12bit ⇒ 2^12 = 4KB

4200 − 4KB

ex1) address 4200 ⇒ code segment ⇒ "01"

13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	1	1	0	1	0	0	0

physical = 104 + 34KB = 34920 B

1. 4200을 표현 ⇒ 4200 − 4KB = 104 = 1101000(2) < 2×2^{10}
2. segment 2bit → 01 ⇒ code segment

ex2) address 7KB = 2^10 * 7 ⇒ 7KB − 4KB = 3KB = 3×2^{10}B = 3072 > 2×2^{10} (segment size보다 큼)

13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	0	0	0	0	0	0	0	0	0

⇒ seg fault 발생 !

- address translation(by hw)

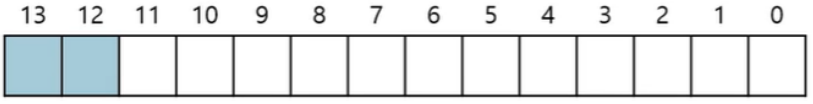
```
// get top 2 bits of 14-bit VA
Segment = (VirtualAddress & SEG_MASK) >> SEG_SHIFT
// now get offset
Offset = VirtualAddress & OFFSET_MASK
if (Offset >= Bounds[Segment])
  RaiseException(PROTECTION_FAULT)
else
  PhysAddr = Base[Segment] + Offset
  Register = AccessMemory(PhysAddr)
```

  - *SEG_MASK* : 0x3000

    → 16진수 → 2^4 ⇒ 한 칸에 4bit를 의미

    - 000 : 12bit를 0으로 채워 넣음
    - 3 : 상위 2bit를 11로 채워 넣음

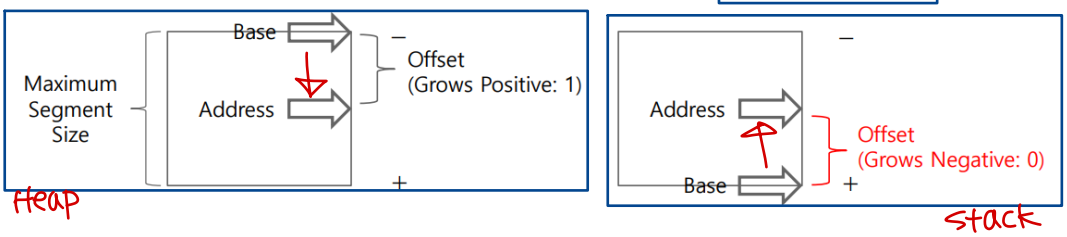
  - *SEG_SHIFT* : 12
    - 12bit만큼 shift
    - 0이면 code, 1이면 heap, 3이면 stack
  - *OFFSET_MASK* : 0xFFF
    - 12bit가 다 1로 채워져 있음



- stack은 어떻게 address translation?
  - stack : 높은 주소가 base , 낮은 쪽으로 계속 나아감
  - hw → segement가 어느 방향으로 계속 배치될 지 알 필요가 있음
    - 가변적으로 메모리가 변할 때 base 주소가 어느 방향으로 변화?
    - segement : 양수 방향은 1로 setting, 음수 방향은 0으로 setting

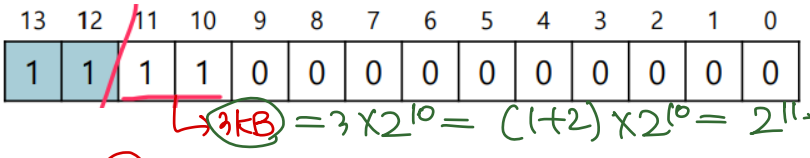
- 해당 값이 없으면 cpu가 translation 못 함

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



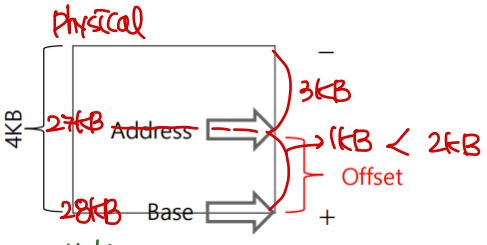
segment마다 base 존재해야 함 → 유효크기가 얼마인지도 표기해야 함

- **Address translation(stack)**
  - example : accessing virtual address 15KB



    - segment : 11 → stack
    - Maximun segment size : 4KB
    - offset : 3KB - 4KB = -1KB (주소가 줄어드는 방향이기에 빼줌)
    - physical address : 28KB - 1KB = 27KB
    - |-1KB| → less than 2KB

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



▼ Support for Sharing

segmentation : physical memory 더 효율적으로 사용할 수 있는 방법

→ segment 사이에 사용되지 않는 공간은 다른 address space를 위해 할당될 수 있음

→ 이 장점을 더 살려서 segment를 별도로 할당하지 않고 공유해서 사용해 보자!

→ (write 빼고 read only인 segment만 공유해보자)

⇒ code segment 공유하기!

- Code Sharing : 더 효율적으로 많은 process 운영하고 싶음
  - code seg : compile 과정에서 다 정해짐. pc가 실행한 코드 부분만 바뀔 뿐

    → 코드 별로 각각 만들 필요가 없음. 다만 코드의 위치(코드 덩어리)만 다를 뿐임
  - to save memory

    → 때때로, address space 사이의 특정 memory segment 공유하기 위해 유용
  - protection bits : 공유하기 위해서 추가적으로 필요한 부분

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

    - read-only였던 code segment를 setting 함으로써

      → 같은 코드는 multiple processess 공유할 수 있음
  - 추가적으로 virtual address가 bound 내에 있는지 check
    - hw : 특정 address의 허가 가능성에 대해서도 check 해야 함

▼ OS support → segmentation을 위해!

- code → segement로 조각냄 ⇒ 해당 조각을 더 조각낸다면?
  - segment 늘려줘야 함
  - base, size pair 또한 늘어나야 함
  - cpu가 그 개수만큼 지원 가능해야 함

- segmentation 어떻게 하는 게 좋을까?
  - **Coarse-grained segmentation → 우리가 지금까지 본 것**
    - 비교적 거대한 segment ⇒ 개수가 그렇게 많이 필요하지는 않음
    - code, stack, heap 수준으로 구분함 ⇒ 꽤 큼
  - **Fine-grained segmentation**
    - 크기가 더 작고 개수가 더 많은 segementation의 필요성

        - 더 효율적으로 사용할 수도 있음
    - 많은 segment 관리하기 위해 segment table이 필요하게 됨

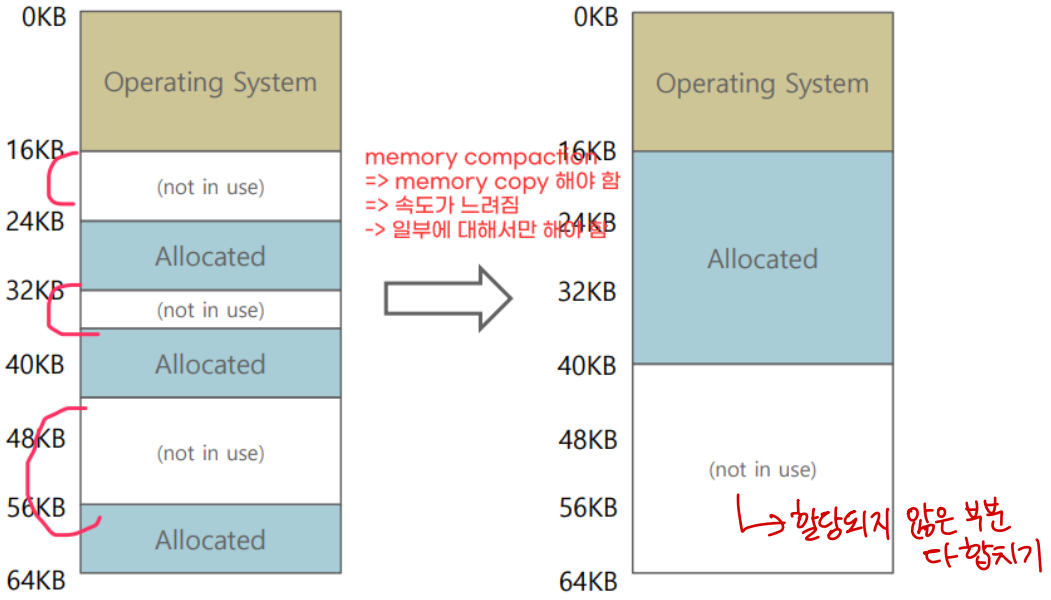
- OS가 support 해줘야 하는 것들
    - Context Switch
        - segment register : 반드시 save, restore해서 update 해줘야 함
            ⇒ OS : 이러한 공간을 효율적으로 사용하기 위한 현실적인 방법이 필요
    - physical memory의 free space 관리하기
        - 새로운 address space가 생성되었을 때
            - OS : 해당 segment를 위한 physical memory 공간을 찾을 수 있어야 함

⇒ 앞으로 이전 예제와 달리 process마다 많은 수의 segment가 있으며 각 segement가 다른 크기를 가지고 있다고 하자

- external frgmentation 발생
    - external frgmentation
        - 남아있는 memory 공간이 process가 요청한 memory 공간보다 큼
        - but, 남아있는 공간이 연속적이지 않아 사용 불가한 경우
    - process마다 종료 시간이 다름
    - 할당되었다가 비는 공간이 불규칙적, 사용 가능한 공간이 쪼개짐 ⇒ 어떻게 활용?

▼ OS support - How?

1. **Physical memory compaction**



24KB → free sapce, 하지만 운영체제는 20KB segment 할당 불가

2. **Free-list management algorithms** ⇒ free list 정렬되는 방법이 각각 다름.

   a. best-fit ← 오름차순
      But. 자투리가 적은 놈들만 많으면 크기 정렬도 애매

      i. 제일 크기가 비슷한 놈부터 → 어디에 넣어도 애매함 ⇒ 크기로 정렬

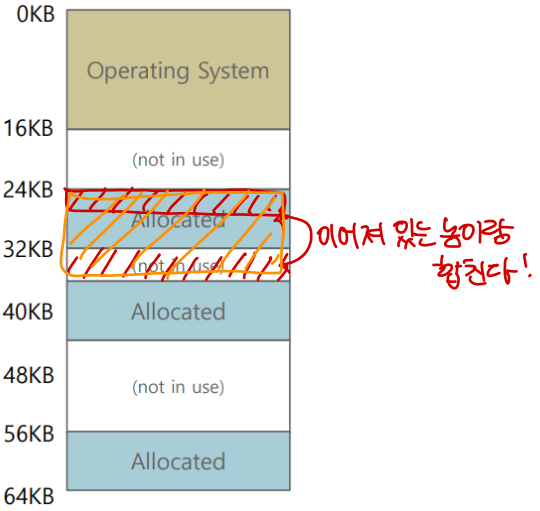
   b. worst-fit → 내림차순.

      i. 제일 크기가 넉넉한 놈부터 → 복불복 ⇒ 크기로 정렬

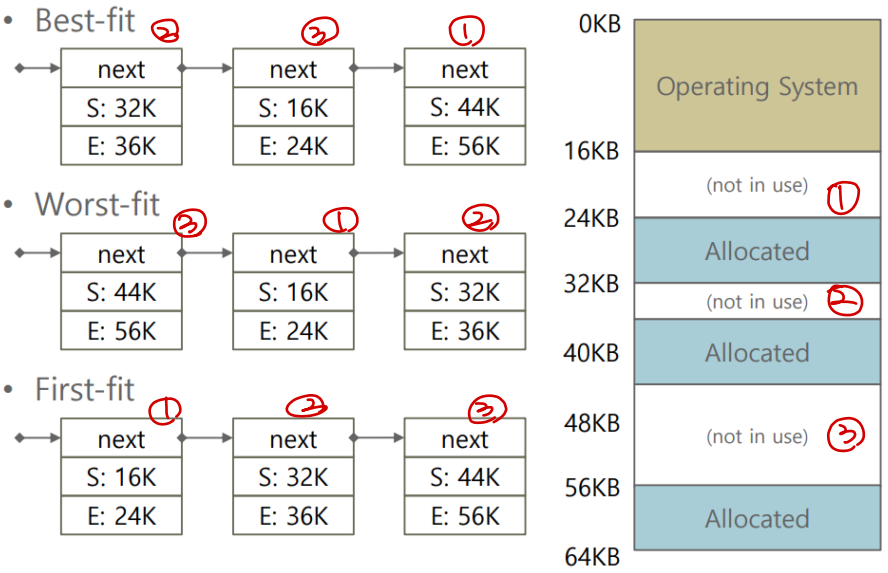
   c. first-fit → 제일 먼저 나오는 놈. (memory copy)

      i. 제일 먼저 찾아진 놈부터 → 관리 쉽게 만들기 ⇒ 주소로 정렬 (address로 정렬함!!)

   d. buddy algorithm



관리하는 free list → 빈공간을 linked list로 관리한다고 가정 ☆



사용자의 응용 sw, 어떤 program 사용.. 하냐에 따라 제일 좋은 것이 달라짐

address → 물리적으로 어떻게 할당?

① Linear
② segment
③ paging