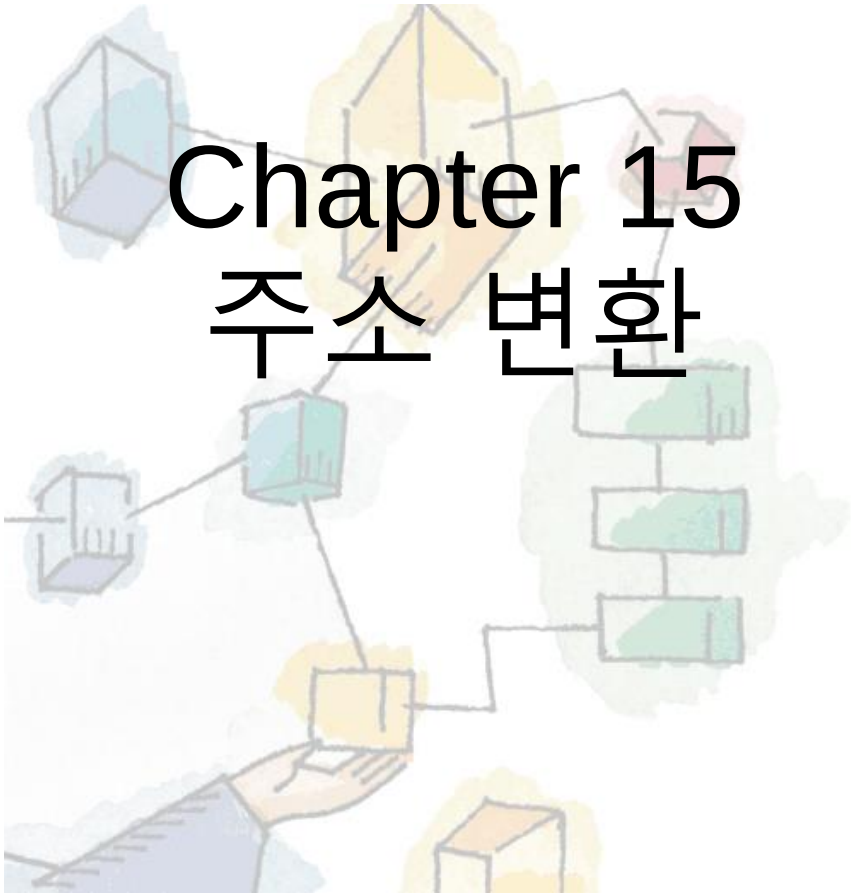
Chapter-15

운영 체제

정내훈

2023년 가을학기
게임공학과
한국공학대학교



Chapter 15
주소 변환

효율적인 메모리 가상화

- 제한적 직접 실행 (Limited Direct Execution)을 사용한 메모리 가상화 구현
  - 직접 실행 : 명령어 대로 CPU가 실행
  - LDE : 일부 명령어의 실행에 OS가 개입
  - OS가 제어권을 얻기위해, 효율적인 구현을 위해 필요
- 제어권 획득과 효율을 위해 하드웨어의 지원이 필요
  - 예) 메모리 관리 레지스터, TLB, 페이지 테이블 등

주소 변환

- **가상 주소**를 **물리 주소**로 변환시키는 것은 하드웨어이다.
  - 실제 정보는 물리주소에 저장된다.
  - 속도가 필요하기 때문, SW로 불가능하기 때문
- OS는 변환에 필요한 중요한 셋업을 수행한다.
  - HW에게 어떻게 변환할지 알려주어야 한다.
  - OS는 메모리를 현명하게 관리해야 한다.

예 : 주소 변환

- 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

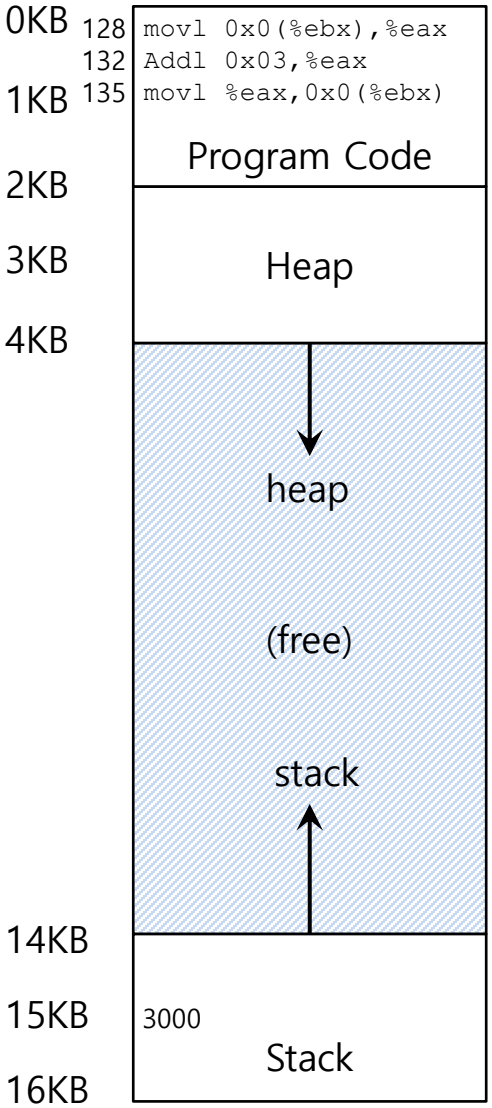
예 : 주소변환 (계속)

- Assembly

```
128 : mov EAX, [EBX]      ; load [ebx] into eax
132 : add EAX, 3          ; add 3 to eax register
135 : mov [EBX], EAX      ; store eax back to mem
```

  - 변수 ‘x’의 주소가 ebx레지스터에 있다고 가정.
  - **Load** : ebx 주소의 메모리 값을 eax레지스터에 복사.
  - **Add** : eax레지스터에 3을 더함.
  - **Store** : eax의 값을 메모리에 다시 복사.

예 : 주소변환 (계속)



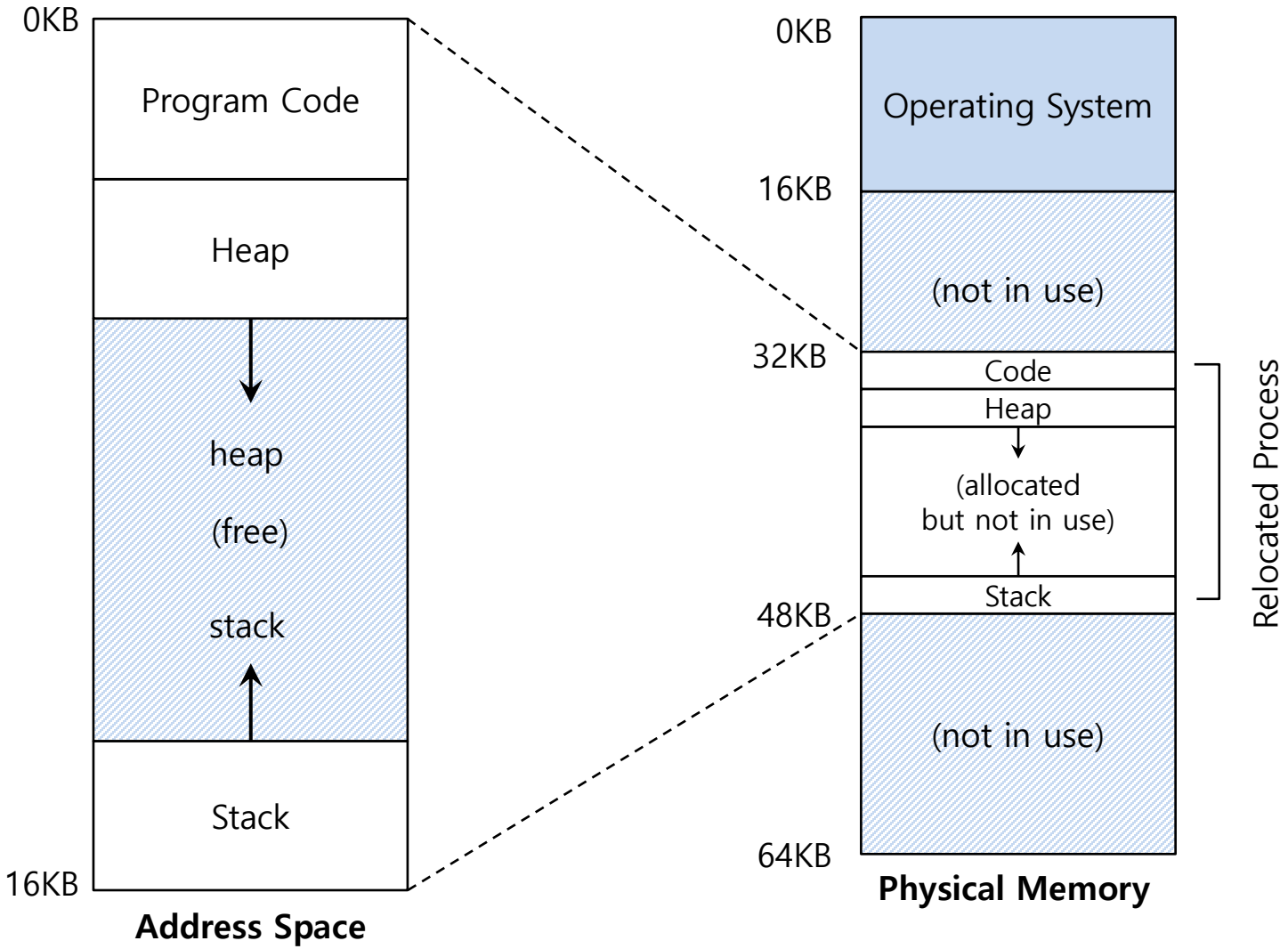
- 주소 128의 명령어 반입(fetch)
- 명령어 실행 (15KB 주소에서 복사)
- 주소 132의 명령어 반입
- 명령어 실행 (메모리 접근 없음)
- 주소 135의 명령어 반입
- 명령어 실행 (15KB 주소에 복사)

모두 5번의 메모리 접근

주소 공간 재배치

- OS는 프로세스를 0번 주소가 아닌 다른 주소에 위치시켜야 한다.
  - 이유? : OS의 위치, 고정된 HW용 자료구조 (메모리의 크기가 가변적이므로)
  - 하지만 컴파일러는 0번 주소에 프로그램을 위치시킨다.
    - 실제 위치를 알 수 없으므로, 컴퓨터마다 OS 버전마다 메모리 크기, 위치가 다르다.
  - 재배치(Relocation)가 필요하다.

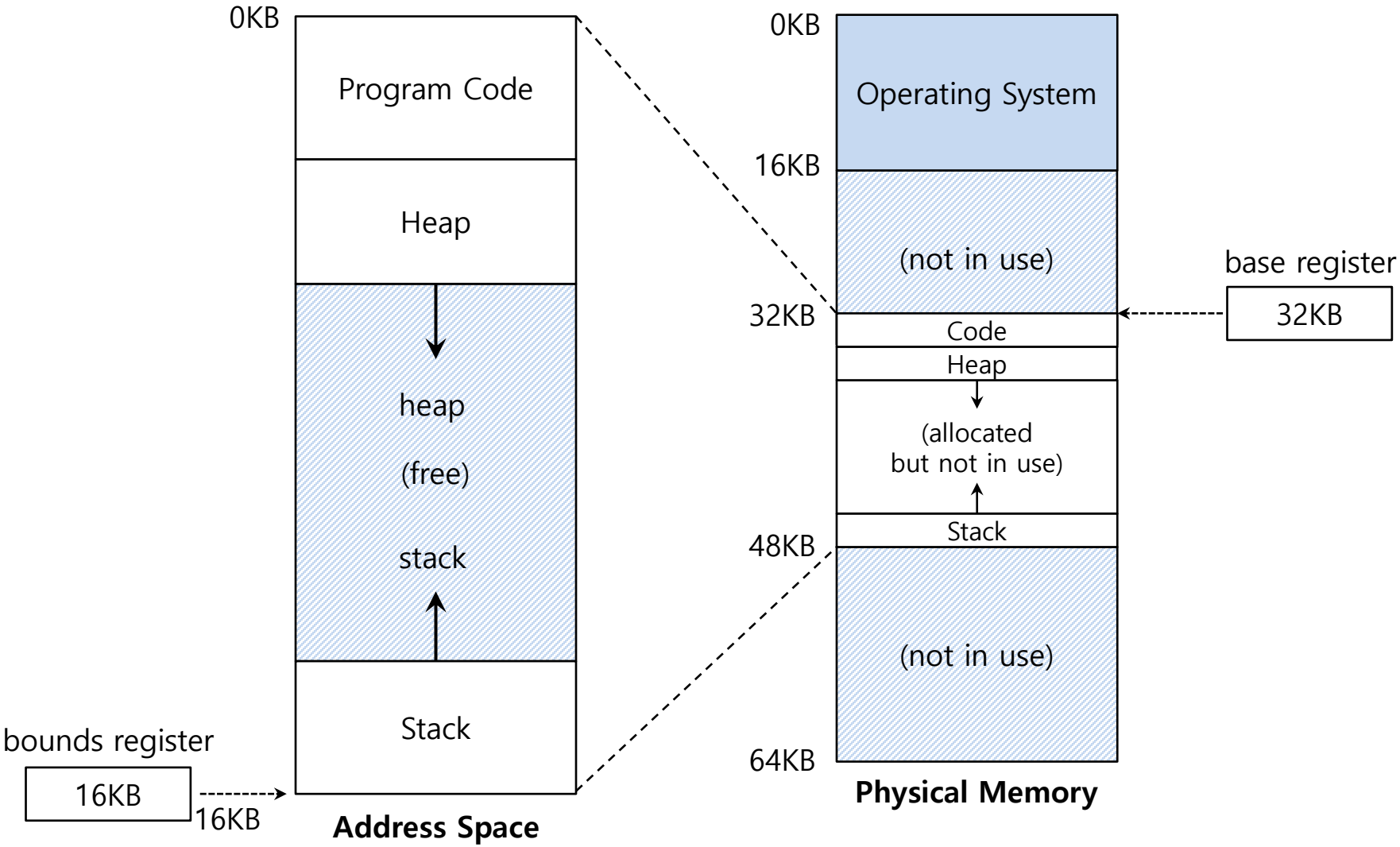
하나의 재배치된 프로세스



동적(하드웨어-기반) 재배치

- HW의 도움이 없는 재배치는 어렵다.
  - 명령어의 모든 주소를 바꿔야 한다.
  - 포인터 변수의 값도 바꿔야 한다.(거의 불가능)
    - 컴파일러의 도움 필요, 프로그램 실행 전에만 가능
  - 정적 재배치라고 불린다.
- 간단한 베이스와 바운드기법에서 세그멘테이션과 페이징으로 진화한다.
- HW 지원
  - 실시간 가상 주소 변환
  - 잘못된 주소 검출 (에러 -> 오류 인터럽트)
  - MMU (Memory Management Unit) 필요

베이스(Base)와 바운드(Bounds) Register



동적 (하드웨어 기반) 재배치

- 프로그램이 시작할 때 OS가 어느 물리주소에 프로그램이 로딩될 지 결정.
  - 베이스 레지스터 설정

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

  - 모든 가상주소는 0보다 크고 바운드값보다 작아야 한다.

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

재배치와 주소변환

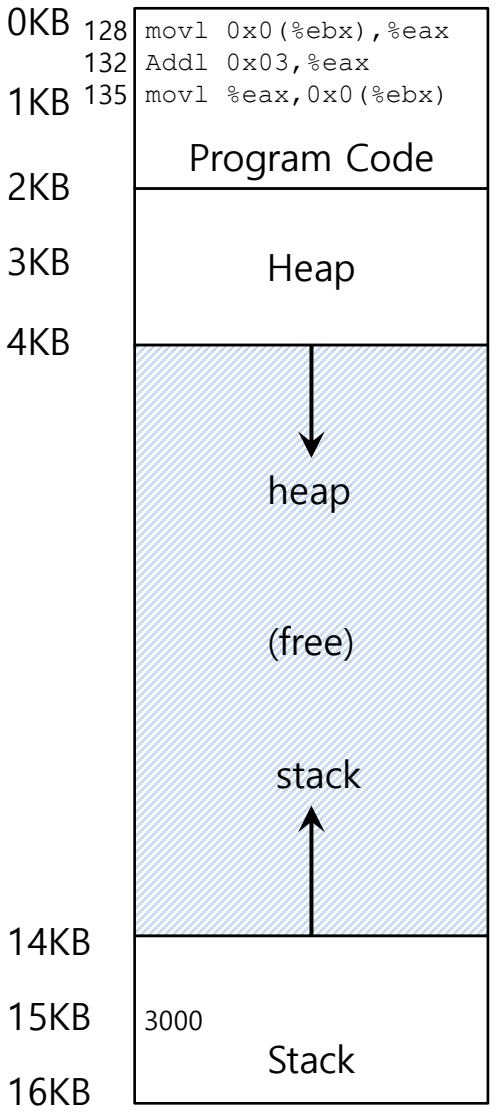
```
128 : movl 0x0(%ebx), %eax
```

- **Fetch:** 주소 128의 명령어 반입

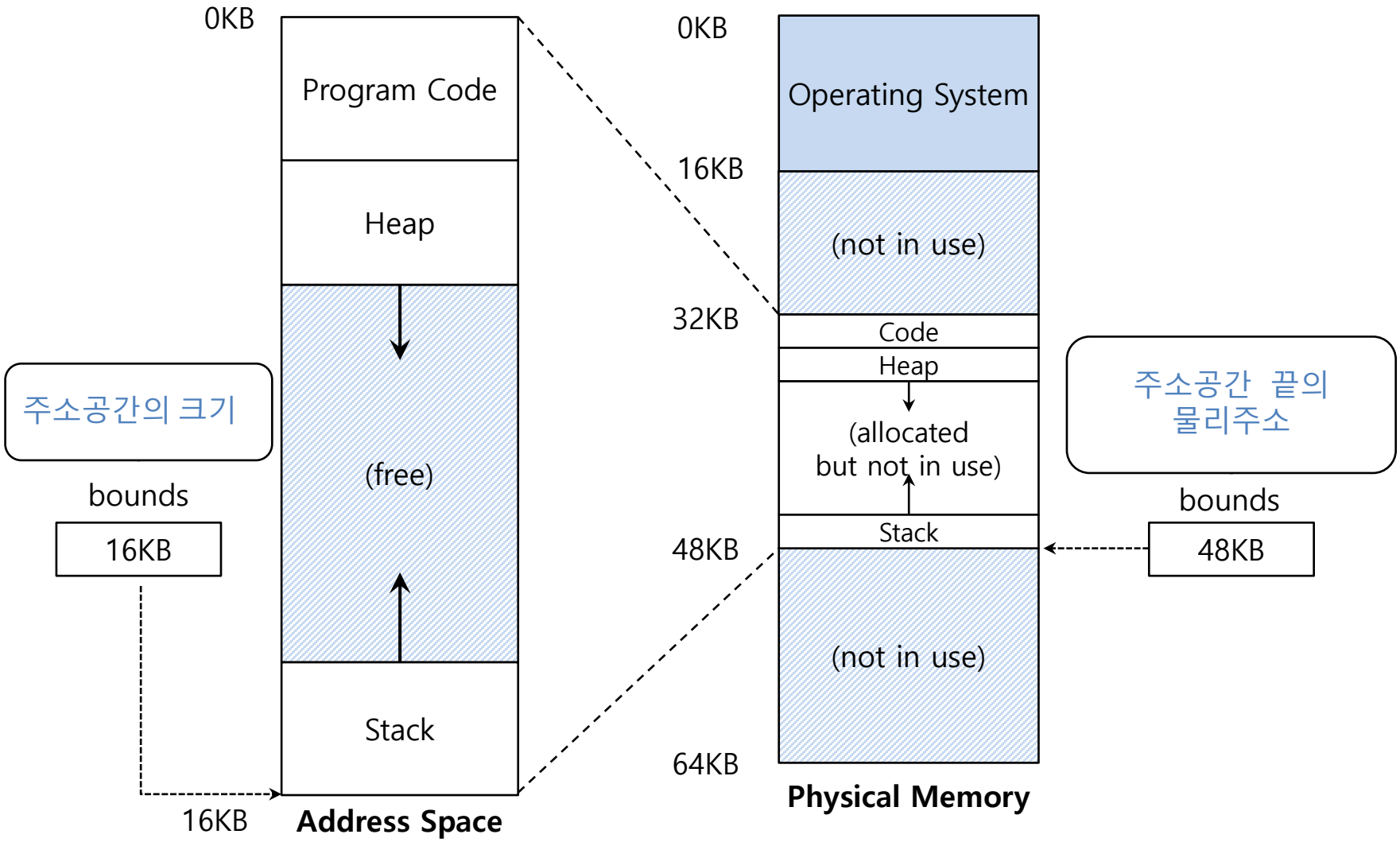
$$32896 = 128 + 32KB(base)$$

- **Execute:** 명령어 실행
  - 15KB 주소에서 복사

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



두 종류의 경계레지스터


0KB
Program Code
Heap
(free)
Stack
16KB
Address Space
주소공간의 크기
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
주소공간 끝의
물리주소
bounds
48KB

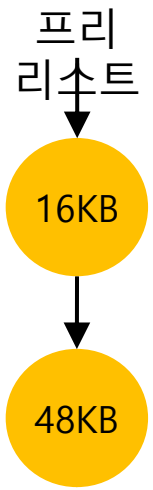
OS 에서의 메모리 가상화

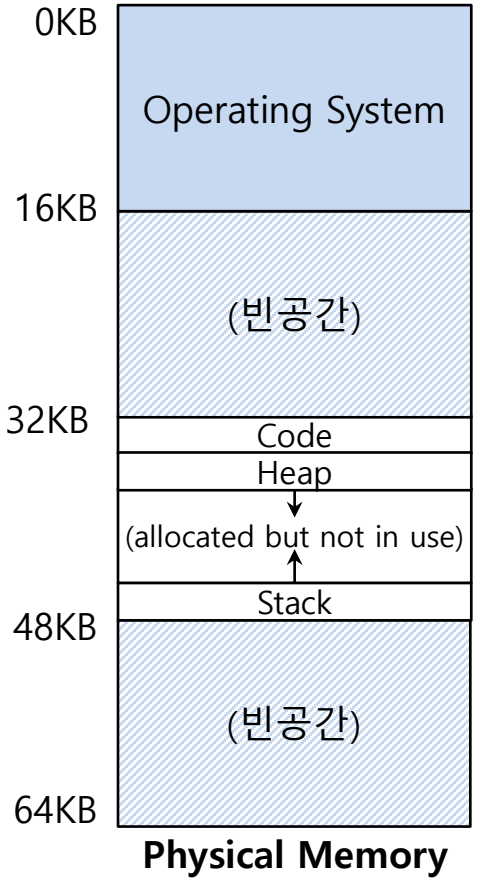
- 베이스와 바운드 기법에서 OS가 할 일
- 3가지 중요한 역할:
  - 프로세스가 시작할 때:
    - 물리 메모리에서 필요한 공간 찾기
  - 프로세스가 종료할 때:
    - 다음 사용자를 위해 메모리 회수
  - 문맥교환이 일어날 때:
    - 베이스와 바운드 레지스터 값 저장
    - PCB에 저장
    - 프로세스의 위치 변경 가능(프로세스 메모리 전체 복사 필요)

OS 측면: 프로세스 시작 시

- OS는 새 주소공간이 차지할 **자리를 찾아야** 한다.
  - 프리 리스트(free list) : 사용하지 않는 물리메모리 공간의 위치를 기록한 리스트.

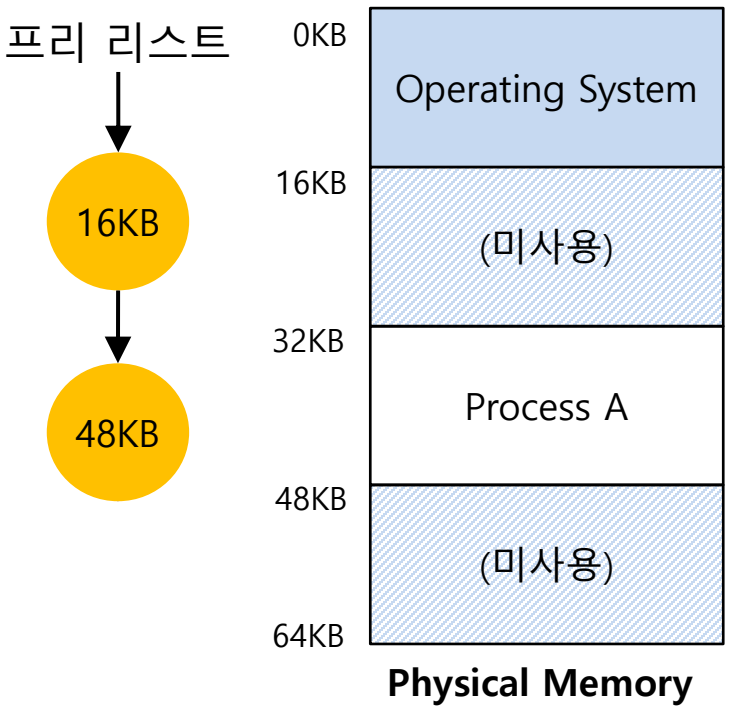
OS는 프리 리스트를 살펴본다.

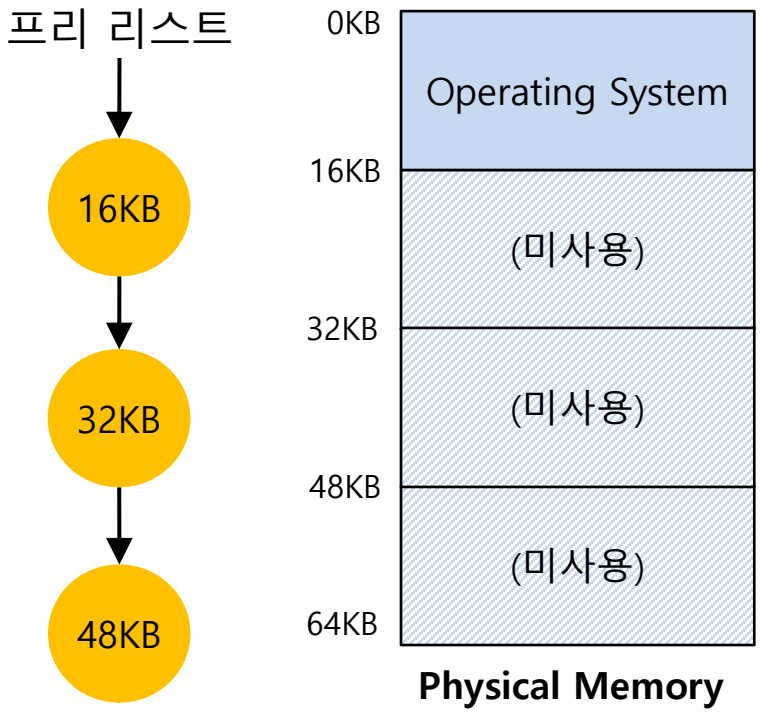




OS 측면 : 프로세스 종료 시

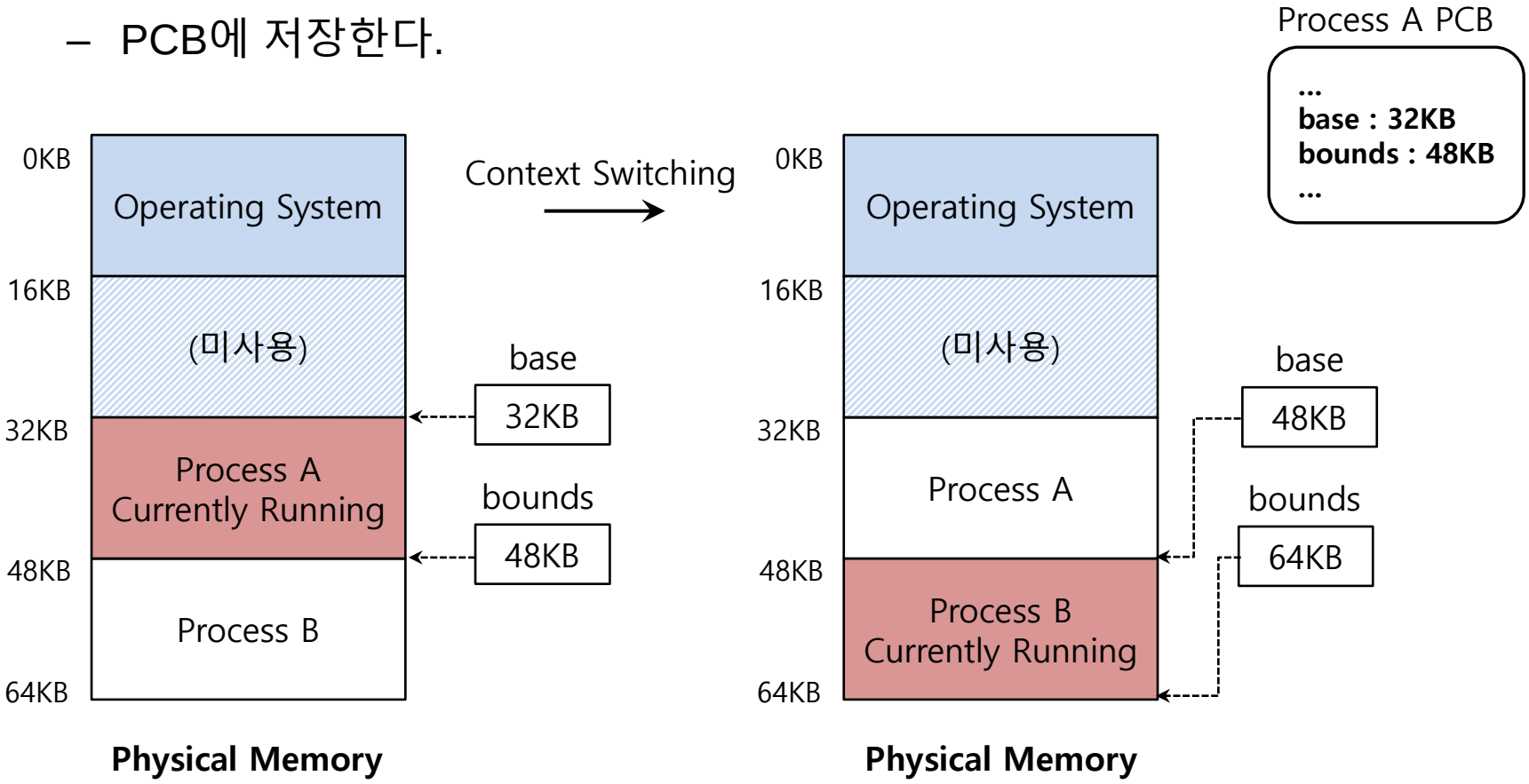
- OS는 할당된 **메모리를 프리리스트에 반환**해야 한다.





OS 측면 : 문맥교환 발생 시

- OS는 베이스와 바운드 레지스터 쌍을 반드시 **저장하고 복원**해야 한다.
  - PCB에 저장한다.



베이스와 바운드의 문제

- 메모리 낭비
  - 내부 공간 낭비 : Heap과 Stack 사이의 공간
    - 내부 단편화 (Internal Fragmentation)