

운영체제

Lecture 08: 메모리 관리



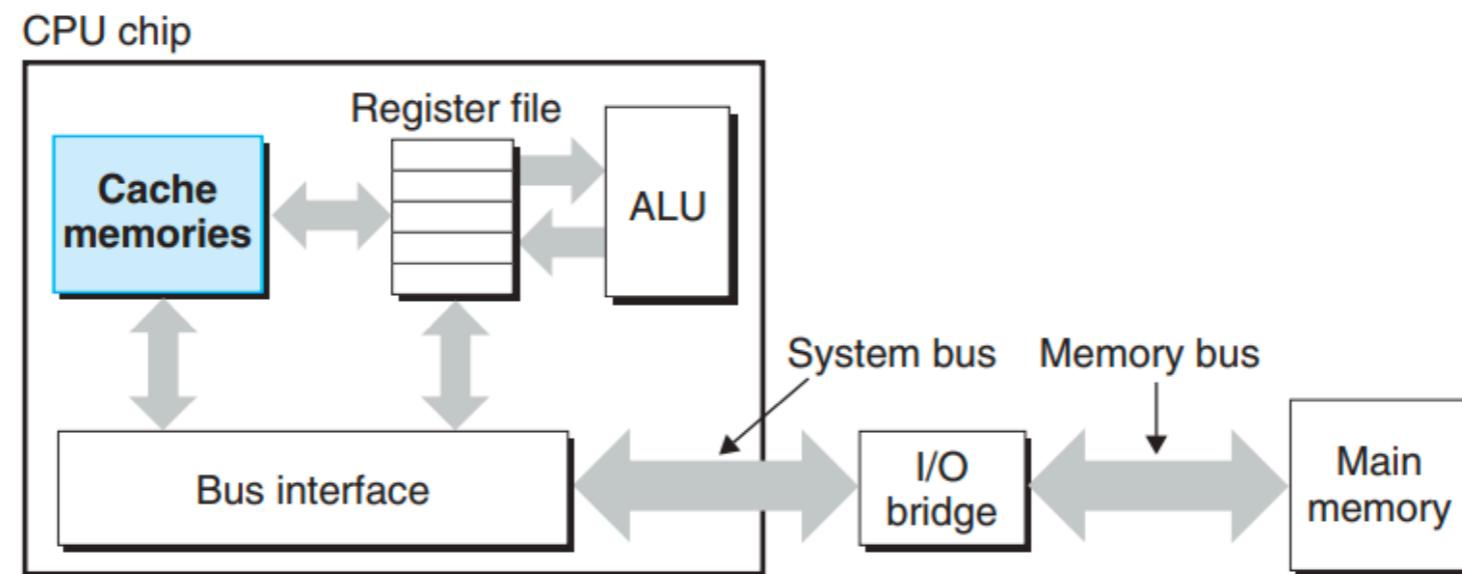
전남대학교 인공지능학부
박태준 (taejune.park@jnu.ac.kr)

지난 시간 복습

- ▶ 교착상태
 - 자원을 소유한 채, 모두 상대방이 소유한 자원을 기다리면서 무한 대기하는 상태
 - 다중 프로그래밍 환경에서는 반드시 발생할 수 밖에 없는 문제..!
- ▶ 교착상태의 네가지 조건
 - 상호 배제 (Mutual Exclusion)
 - 비선점 (No Preemption)
 - 점유와 대기 (Hold & Wait)
 - 환형 대기(Circular Wait)
- ▶ 교착상태의 회복
 - 예방(prevention), 회피(avoidance), 감지 및 복구(detection and recovery), 무시

지금 까지 배운 내용 정리

- ▶ 지금까지 배운 내용 정리...!



- 주로 CPU관점에서 봤다고 볼 수 있음
 - CPU는 메모리에 프로그램이 올라와야 읽고 실행할 수 있다!
 - 메모리에 올라온 프로그램? → 프로세스!
 - OS는 프로세스를 어떻게 관리하는가? PCB
 - 프로세스의 라이프 사이클, 쓰레드, 스케줄링,
 - IPC, 그리고 공유자원 문제, 교착상태
- ▶ 좋아요! 이제 OS가 프로그램을 어떻게 관리하는지는 알았어요
그러면 그 관리하는 장소는 어떻게 관리할까요? → **Memory**

Goal

- ▶ 메모리 계층 구조에 대한 이해! (배웠던 것)
- ▶ 메모리 물리주소와 논리 주소 (feat. 가상메모리)
 - 프로세스때 살짝 다루었던 이야기.
- ▶ 프로세스의 실행에 필요한 메모리 할당 정책에 대한 이해, 메모리 할당 알고리즘
 - 연속 메모리 할당
 - 분할 메모리 할당 (이건 다음 챕터에서...!)
 - first-fit, best-fit, worst-fit
 - 그리고 단편화
- ▶ Segmentation, Buddy system

메모리 관리

메모리 관리!

▶ 운영체제에 의해 메모리 관리가 필요한 이유

- **메모리는 공유 자원**

- 여러 프로세스 사이에 메모리 공유
- 각 프로세스에게 물리 메모리 할당

- **메모리 보호**

- 프로세스의 독립된 메모리 공간 보장
- 다른 프로세스로부터 보호
- 사용자 코드로부터 커널 공간 보호

- **메모리 용량 한계 극복**

- 설치된 물리 메모리보다 큰 프로세스 지원 필요
- 여러 프로세스의 메모리 합이 설치된 물리 메모리보다 큰 경우 필요

- **메모리 효율성 증대**

- 가능하면 많은 개수를 프로세스를 실행시키기 위해
- 프로세스당 최소한의 메모리 할당

일단 메모리 간단히 복습...!

- ▶ 메모리는 컴퓨터 시스템 여러 곳에 계층적으로 존재
 - CPU 레지스터 – CPU 캐시 – 메인 메모리 – 보조기억장치
 - CPU 레지스터에서 보조기억장치로 갈수록
 - 용량 증가, 가격 저렴, 속도 저하
 - 메모리 계층 구조의 중심 – 메인 메모리
- ▶ 메모리 계층화의 목적
 - CPU의 메모리 액세스 시간을 줄이기 위함 ⇒ 빠른 프로그램 실행

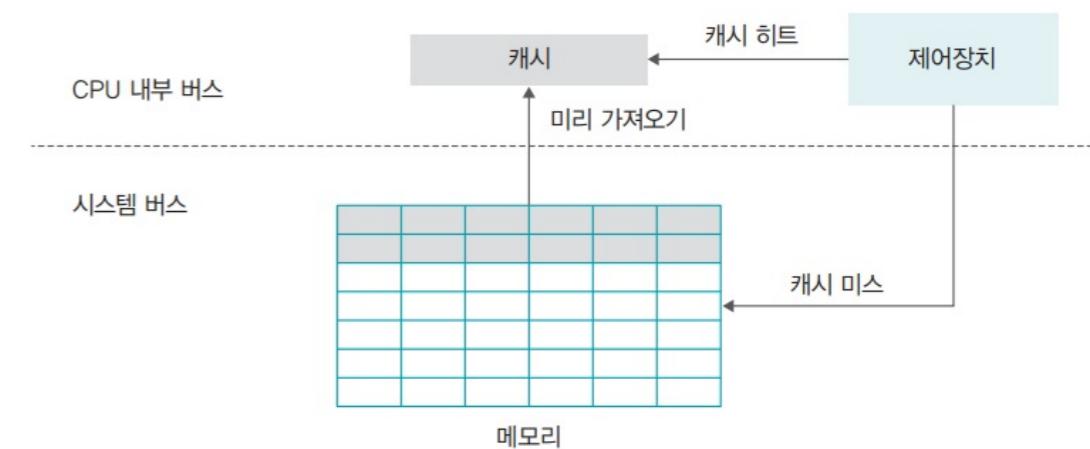
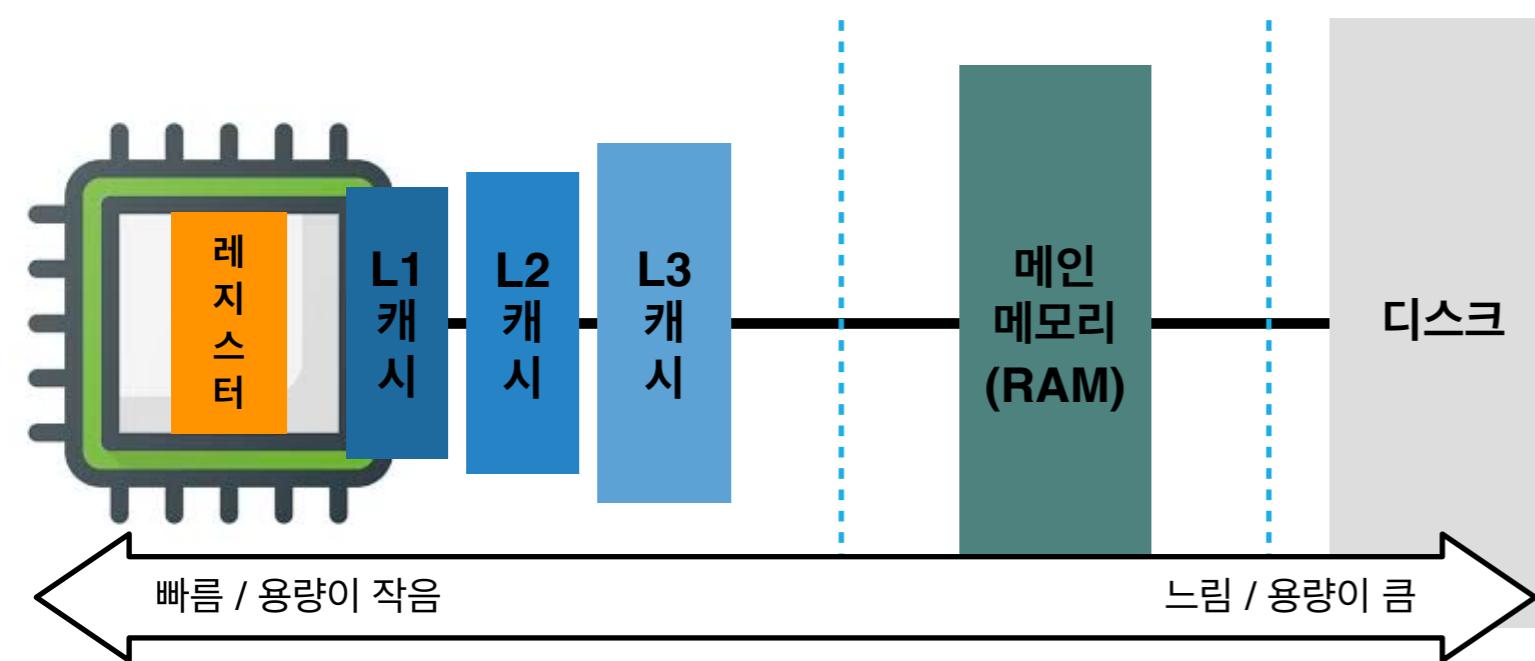
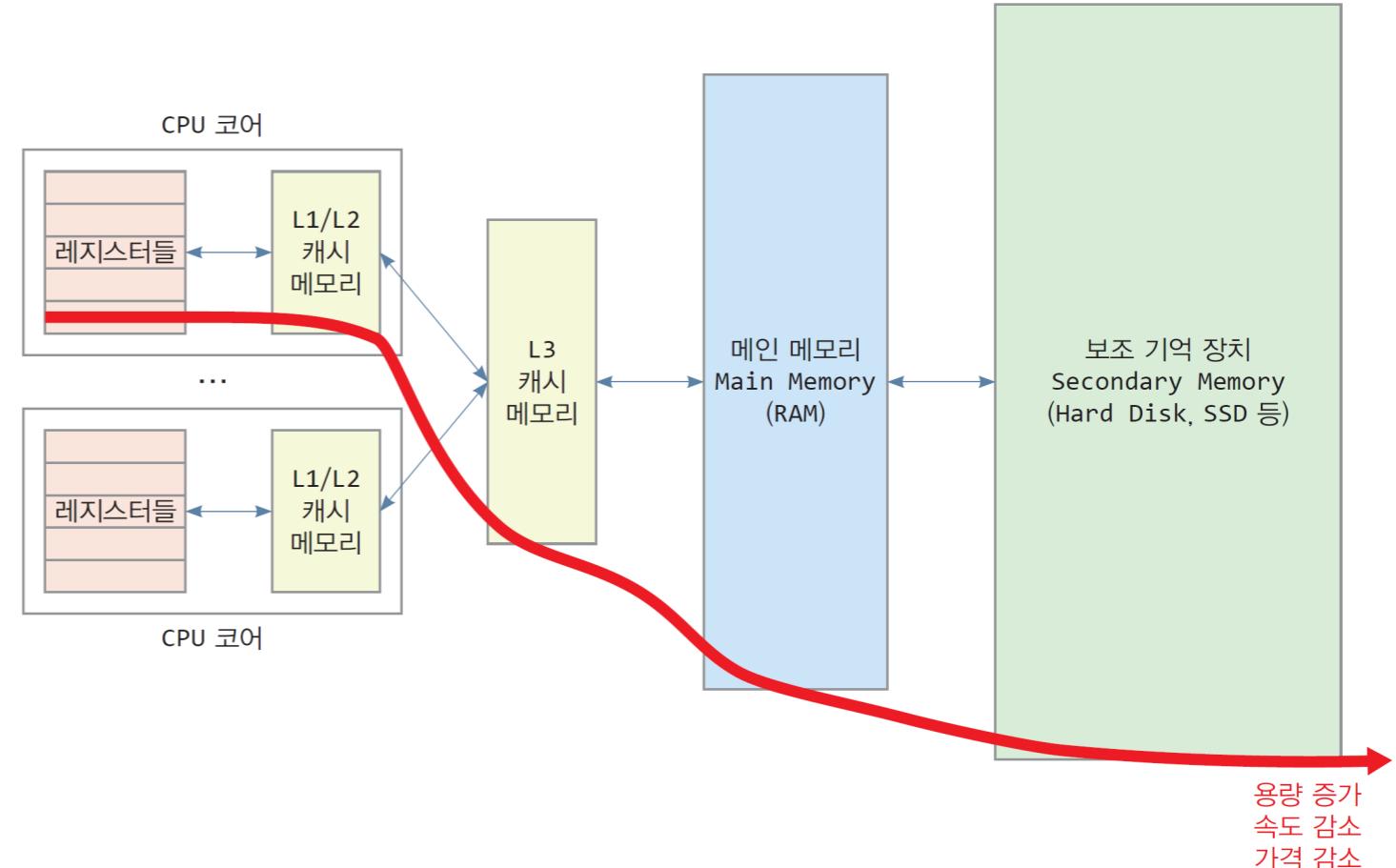


그림 2-18 캐시의 구조

메모리 계층 구조

▶ 계층화가 가능한 이유?!

- 참조의 지역성 때문 Locality
- 코드나 데이터, 자원 등이 아주 짧은 시간 내에 다시 사용되는 프로그램의 특성
 - CPU는 작은 캐시 메모리에 로딩된 코드와 데이터로 한동안 실행
 - 캐시를 채우는 시간의 손해보다 빠른 캐시를 이용하는 이득이 큼



▶ 시간 지역성 (Loop)

▶ 공간 지역성 (Array)

	CPU 레지스터	L1/L2 캐시	L3 캐시	메인 메모리	보조 기억 장치
용도	몇 개의 명령과 데이터 저장	한 코어에서 실행되는 명령과 데이터 저장	멀티 코어들에 의해 공유. 명령과 데이터 저장	실행 중인 전체 프로세스들의 코드와 데이터, 입출력 중인 파일 블록들 저장	파일이나 데이터베이스, 그리고 메모리에 적재된 프로세스의 코드와 데이터의 일시 저장
용량	바이트 단위. 8~30개 정도. 1KB 미만	KB 단위 (Core i7의 경우 32KB/256KB)	MB 단위 (Core i7의 경우 8MB)	GB 단위 (최근 PC의 경우 최소 8GB 이상)	TB 단위
타입		SRAM (Static RAM)	SRAM (Static RAM)	DRAM (Dynamic RAM)	마그네틱 필드나 플래시 메모리
속도	<1ns	<5ns	<5ns	<50ns	<20ms
가격		고가	고가	보통	저가
휘발성	휘발성	휘발성	휘발성	휘발성	비휘발성

지역성, 눈으로 보자!

```
#include <stdio.h>

#define ARR_SIZE 1024

void func() {
    int arr[ARR_SIZE][ARR_SIZE] = {0,};

    for (int i = 0; i < ARR_SIZE; i++)
        for (int j = 0; j < ARR_SIZE; j++) {
            arr[i][j] = 1;
            //arr[j][i] = 1;
        }
}

int main() {
    for (int i = 0 ; i < 100; i++)
        func();
}
```

그대를 위한 복불 (for xNIX)

arr[**i**][**j**] = 1 의 경우

```
taejune@Taejunes-MBP2021 memory % time ./a.out
./a.out 0.12s user 0.00s system 98% cpu 0.129 total
taejune@Taejunes-MBP2021 memory %
```

arr[**j**][**i**] = 1 의 경우

```
taejune@Taejunes-MBP2021 memory % time ./a.out
./a.out 0.35s user 0.00s system 99% cpu 0.359 total
taejune@Taejunes-MBP2021 memory %
```

코드 한줄 다르게 적었다고 성능이 2배!

메모리 주소와 CPU bit

▶ 메모리 주소

- 1Byte로 나눈 메모리의 각 영역은 메모리 주소로 구분. 0번지부터 시작
- CPU는 메모리에 있는 내용을 가져오거나 작업 결과를 메모리에 저장하기 위해 메모리 주소 레지스터(MAR)를 사용

▶ CPU의 비트

- 한 번에 다룰 수 있는 데이터의 최대 크기 ~ 워드(word)!
- 32bit CPU는 한 번에 다룰 수 있는 데이터의 최대 크기가 32bit
 - 1word = 32bit
 - 표현할 수 있는 메모리 주소의 범위가 $0 \sim 2^{32}-1$
 - 16진수로 나타내면 00000000~FFFFFF, 총크기는 약 4GB
- 64bit CPU 내의 레지스터 크기는 전부 64bit,
산술 논리 연산장치와 대역폭도 64bit
 - $0 \sim 2^{64}-1$ 번지의 주소 공간을 제공, 약 16,777,216TB

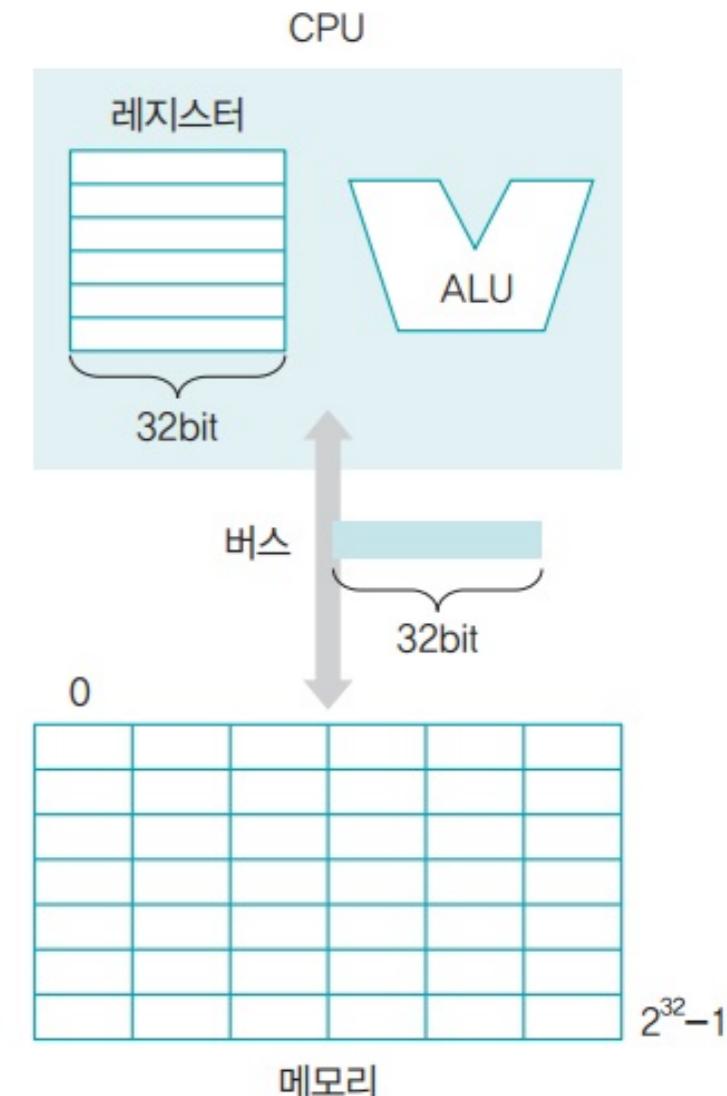


그림 7-8 32bit CPU의 구조

메모리 관리는 왜 어려운가...!?

- ▶ 메모리는... 유한하다! 그런데 한없이 부족하다!
- ▶ 폰노이만 구조 → 메모리는 컴퓨터의 유일한 작업공간. 모든 프로그램은 메모리에!
 - 일괄처리 시스템: 한번에 하나의 프로그램만 실행시킴으로 메모리 관리가 단순함
 - 다중 프로그래밍: 운영체제를 포함한 모든 응용 프로그램이 메모리에 올라와야 함.
- ▶ 같이 쓰다 보니...
 - 여러 프로세스간 침범이 안되게 하면서,
 - 그러면서 필요한 공간은 충분히 할당해줘야 하고...
 - 가능한 많은 프로세스를 동작시켜야 하는데, 메모리는 부족하고...
- ▶ 메모리 관리의 이중성
 - 프로세스: 가능한 많은 메모리를 쓰고 싶어함 (메모리 독점)
 - OS / 관리자: 가능한 메모리를 적게 주고 싶어함. 가능한 효율적으로 사용하고 싶어함.
- ▶ 체감적으로 성능과 가장 직결되는 부분.

그래서 뭘 어떻게 한단 말인가?

- ▶ 컴퓨터의 메모리를 어떻게 관리 할 것인가? → 메모리 관리자의 작업 / 정책
 - 적재 정책 (**Fetch policy**)
 - 프로세스가 필요로 하는 데이터를 언제 메모리로 가져올지 결정하는 정책
 - 요구반입 vs 예상반입
 - 배치 정책 (**Placement policy**)
 - 가져온 프로세스를 메모리의 어떤 위치에 올려놓을지 결정하는 정책
 - 최초적합 vs 최적적합 vs 최악적합
 - 재배치(교체) 정책 (**Replacement policy**)
 - 메모리가 꽉 찼을 때 메모리 내에 있는 어떤 프로세스를 내보낼지 결정하는 정책
 - 최적교체, FIFO, LRU(Least Recently Used), LFU(Least Frequently Used)
- ▶ 일반적으로 MMU가 메모리 관리자라 불림

메모리 주소

물리 주소와 논리 주소

- ▶ 메모리는 오직 주소로만 접근
- ▶ 물리 주소(physical address) ~ 하드웨어 관점
 - 물리 메모리(RAM)에 매겨진 주소, 하드웨어에 의해 고정된 메모리 주소
 - 0에서 시작하여 연속되는 주소 체계
 - 메모리는 시스템 주소 버스를 통해 물리 주소의 신호 받음
- ▶ 논리/가상 주소(logical address/virtual address) ~ 프로세스 관점
 - 개발자나 프로세스가, 프로세스 내에서 사용하는 주소, 코드나 변수 등에 대한 주소
 - CPU가 프로세스를 실행하는 동안 다루는 모든 주소는 논리 주소
 - 프로세스 내에서 매겨진 **상대 주소** ~ 0에서 시작하여 연속되는 주소 체계
 - 프로그램에서 변수 n의 주소가 100번지라면, 논리 주소가 100이고, 물리 주소를 알 수 없음, 실제 메모리의 주소 아님
 - 컴파일러와 링커에 의해 매겨진 주소
 - 실행 파일에 내에 만들어진 이진 프로그램의 주소들은 논리 주소로 되어 있음
 - 사용자나 프로세스는 결코 물리 주소를 알 수 없음
- ▶ 메모리 접근 시 상대 주소를 사용하면 절대 주소로 변환해야 함 → MMU



(a) 절대 주소

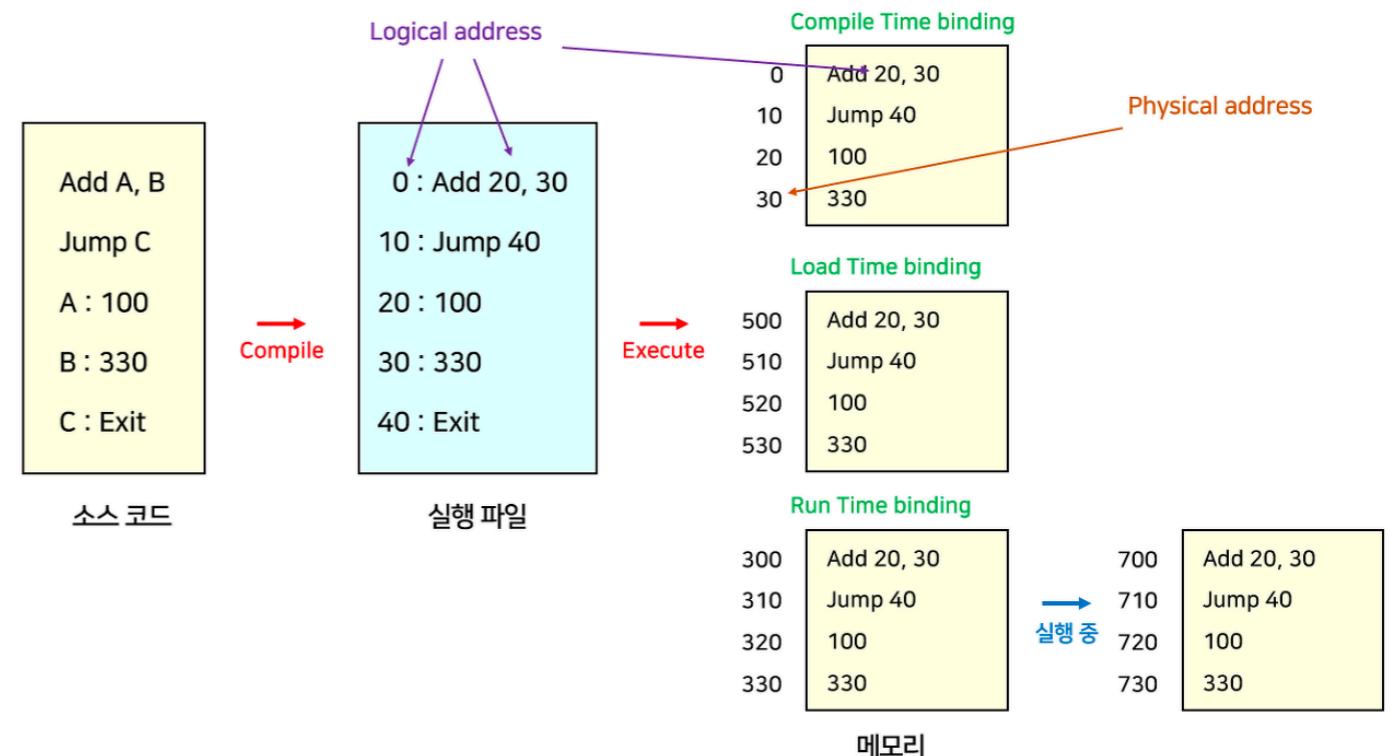


(b) 상대 주소

그림 7-12 절대 주소와 상대 주소

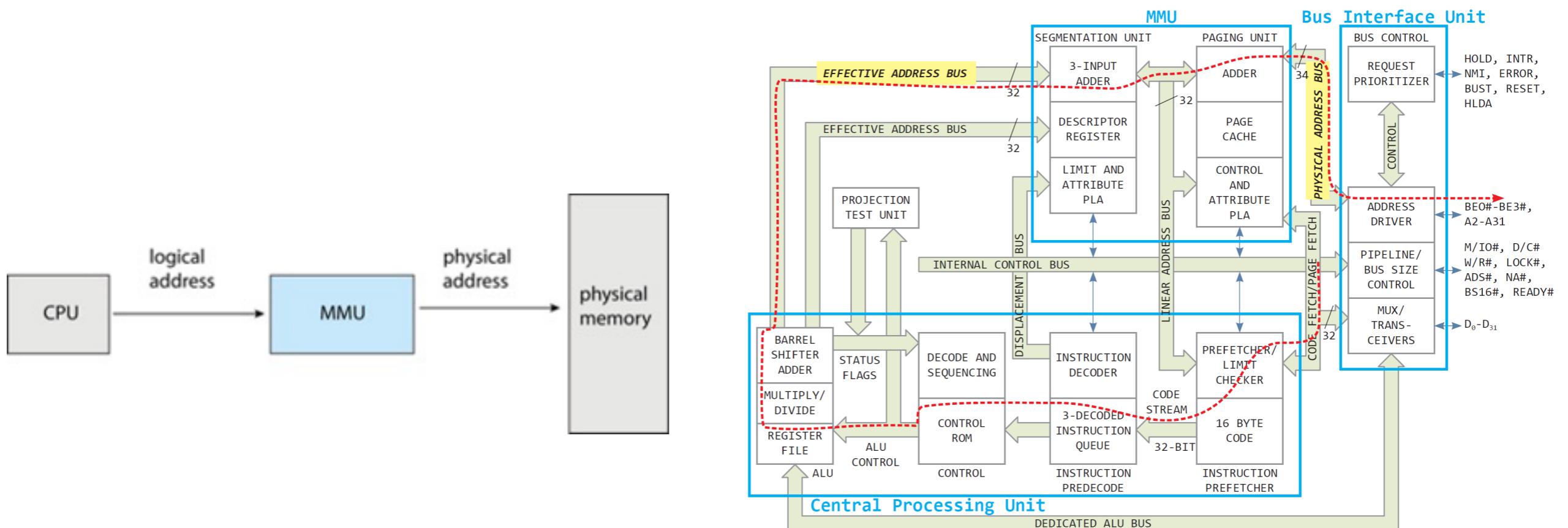
무슨 말인가? ~ Address binding의 개념

- ▶ 프로그램이 메모리 어느 위치에, 즉 어떤 물리적 위치에 Load될지 결정하는 과정
 - **Compile-time binding:** 프로세스의 물리적 주소가 컴파일 타임에 정해짐
 - 물리주소 == 논리주소 → 같은 주소를 쓰는 프로세스가 있다면?? 적재 불가능. 매우 비효율
 - **Load-time binding:** Loader가 프로세스를 load 하는 시점에 물리적 주소 결정 (Relocatable)
 - 바꿔야할 주소가 많다면? 로딩에 부담!
 - **Run-time binding:** 프로세스가 수행이 시작된 이후에 프로세스가 실행될 때 메모리 주소를 바꿈
 - Runtime때 물리적 주소가 결정
 - **MMU(Memory Management Unit)**



MMU (Memory Management Unit)

- ▶ 논리 주소를 물리 주소로 바꾸는 하드웨어 장치
 - CPU가 발생시킨 논리 주소는 MMU에 의해 물리 주소로 바뀌어 메모리에 도달
- ▶ 오늘날 MMU는 CPU 안에 내장
 - 인텔이나 AMD의 x86 CPU는 80286부터 MMU를 내장
 - MMU 덕분으로 여러 프로세스가 하나의 메모리에서 실행되도록 됨



상대주소 → 물리주소

- 메모리 관리자는 사용자 프로세스가 상대 주소를 사용하여 메모리에 접근할 때마다 상대 주소값에 Base 레지스터 값을 더하여 절대 주소를 구함
 - Base 레지스터는 주소 변환의 기본이 되는 주소값을 가진 레지스터로, 메모리에서 사용자 영역의 시작 주소값이 저장 (접근할 수 있는 물리적 주소의 최소값)
 - Limit 레지스터는 메모리 관리자는 사용자가 작업을 요청할 때마다 경계 (상한)을 벗어나는 작업을 요청하는 프로세스가 있으면 그 프로세스를 종료

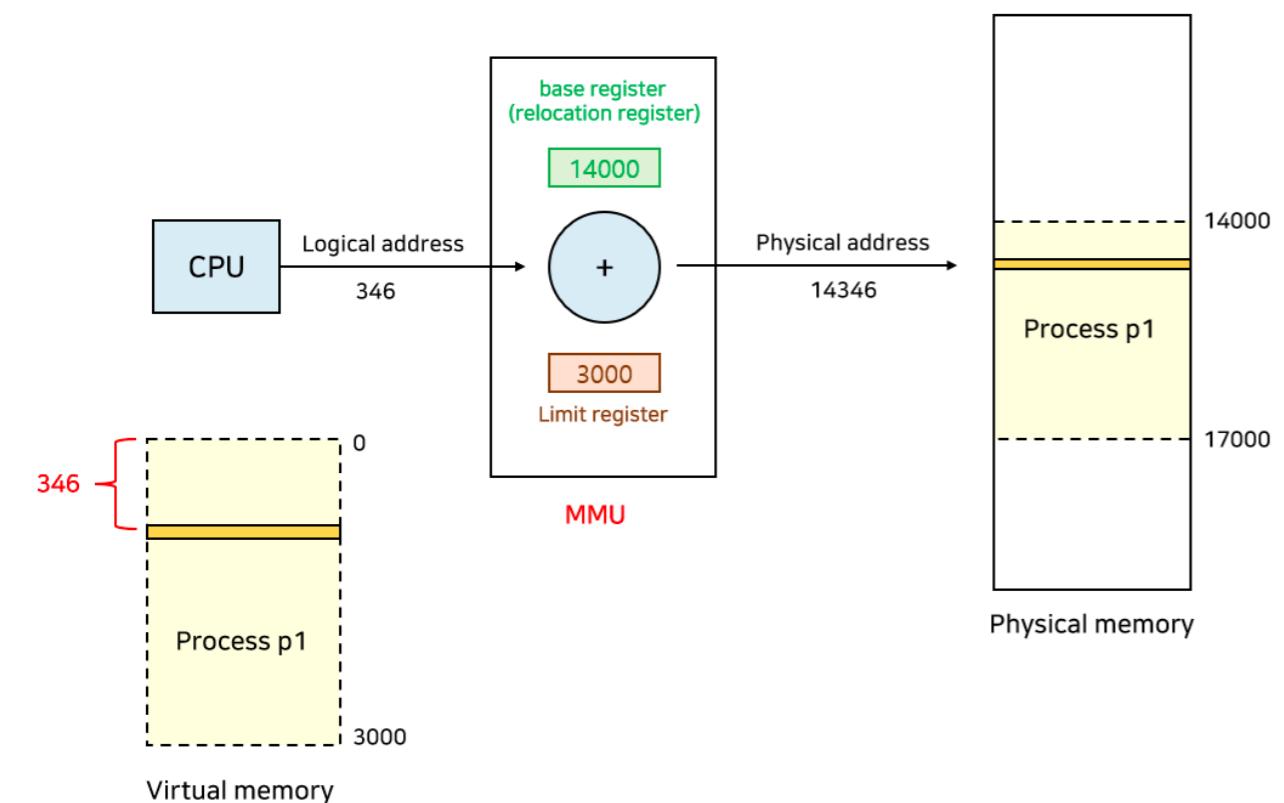
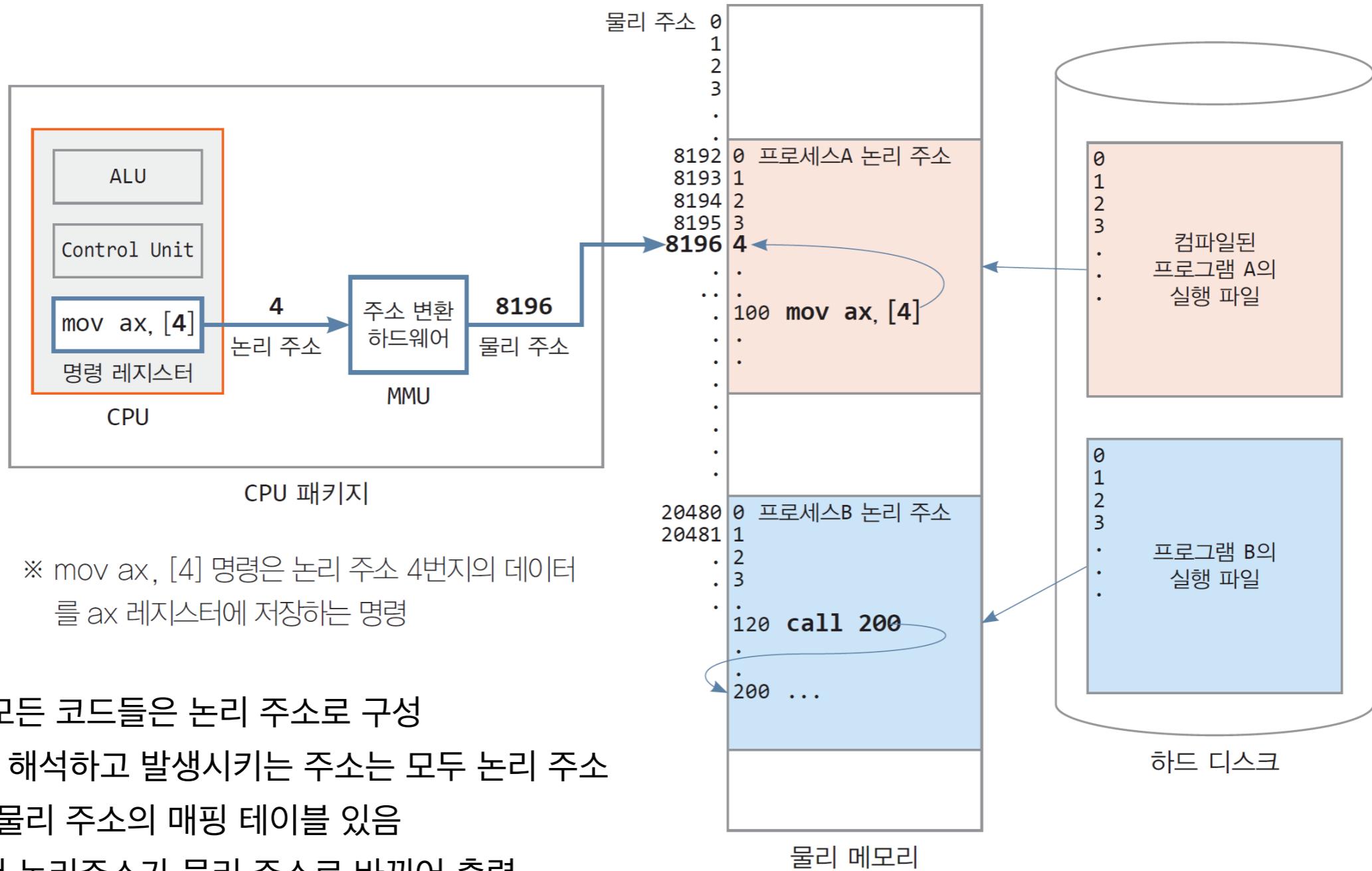


표 7-3 절대 주소와 상대 주소

구분	절대 주소	상대 주소
관점	메모리 관리자 입장	사용자 프로세스 입장
주소 시작	물리 주소 0번지부터 시작	물리 주소와 관계없이 항상 0번지부터 시작
주소 공간	물리 주소(실제 주소) 공간	논리 주소 공간

주소 변환

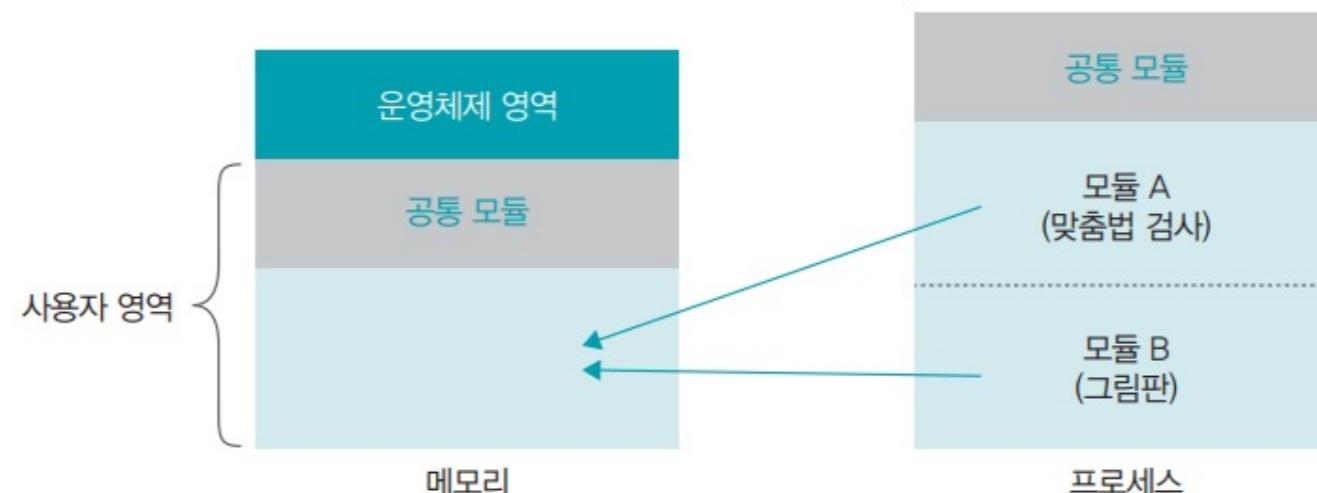


메모리 오버레이

- ▶ **가상 메모리 공간:** 각 프로세스는 자신이 모든 메모리를 점유하고 있는 것 처럼 생각한다.
 - 실제 할당된 공간은 작음! → 프로세스가 실제 필요한 공간보다 물리공간이 더 작을 수 있음.

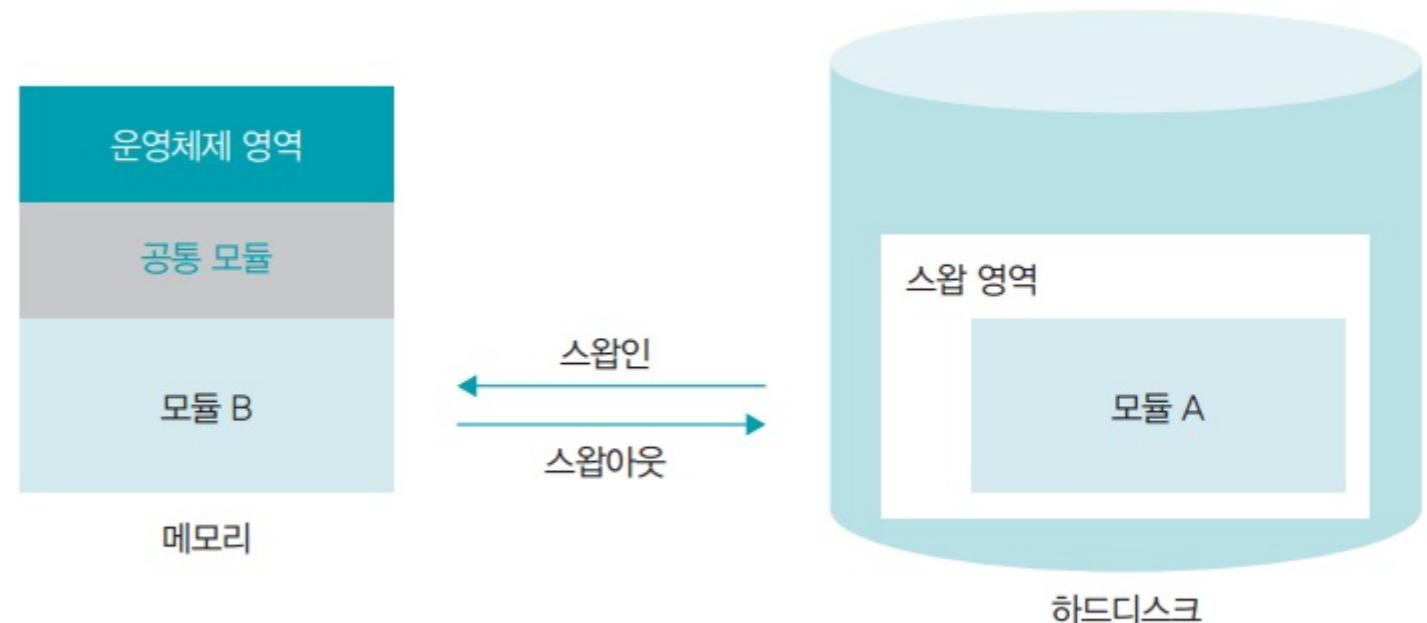
▶ 메모리 오버레이

- 프로그램의 크기가 실제 메모리(물리 메모리)보다 클 때 전체 프로그램을 메모리에 가져오는 대신 적당한 크기로 잘라서 가져오는 기법
- **프로그램이 실행되면 현재 필요한 부분만 메모리에 올라와 실행 → 바꿔가면서 (Swap)**
 - 한정된 메모리에서 메모리보다 큰 프로그램 실행 가능
 - 프로그램 전체가 아니라 일부만 메모리에 올라와도 실행 가능



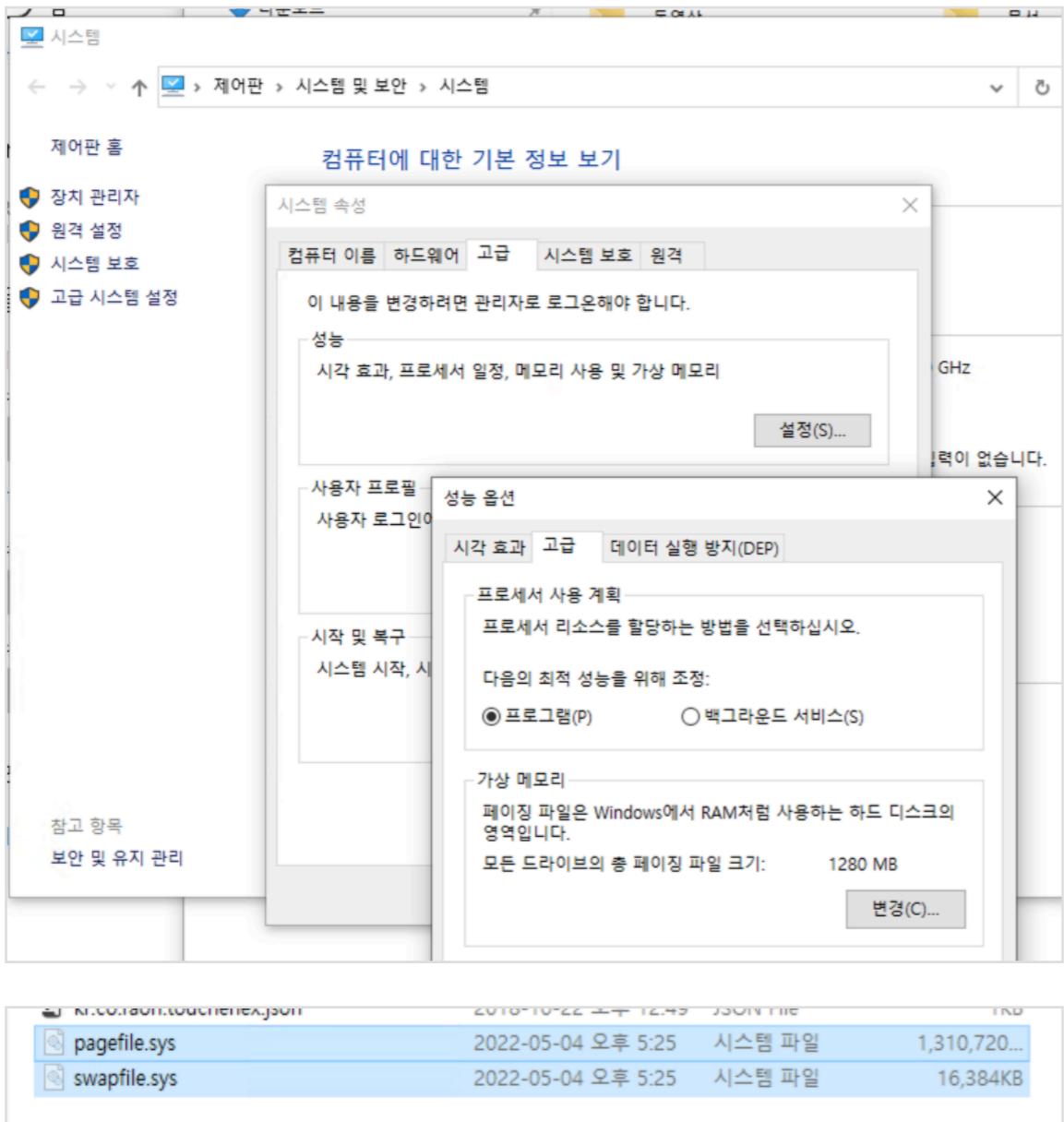
Swap

- ▶ Swap area
 - 메모리가 모자라서 쫓겨난 프로세스를 저장장치의 특별한 공간에 모아두는 영역
 - 메모리에서 쫓겨났다가 다시 돌아가는 데이터가 머무는 곳이기 때문에 저장장치는 장소만 빌려주고 메모리 관리자가 관리
 - 사용자는 실제 메모리의 크기와 스왑 영역의 크기를 합쳐서 전체 메모리로 인식하고 사용
- ▶ Swap in, swap out
- ▶ 들락날락 하다보면....?
 - 단편화 문제 발생...!
 - 성능 문제!



여러분 컴퓨터에서!

- ▶ 윈도우에서는 이러한 것을 '가상메모리'라고 표현합니다.

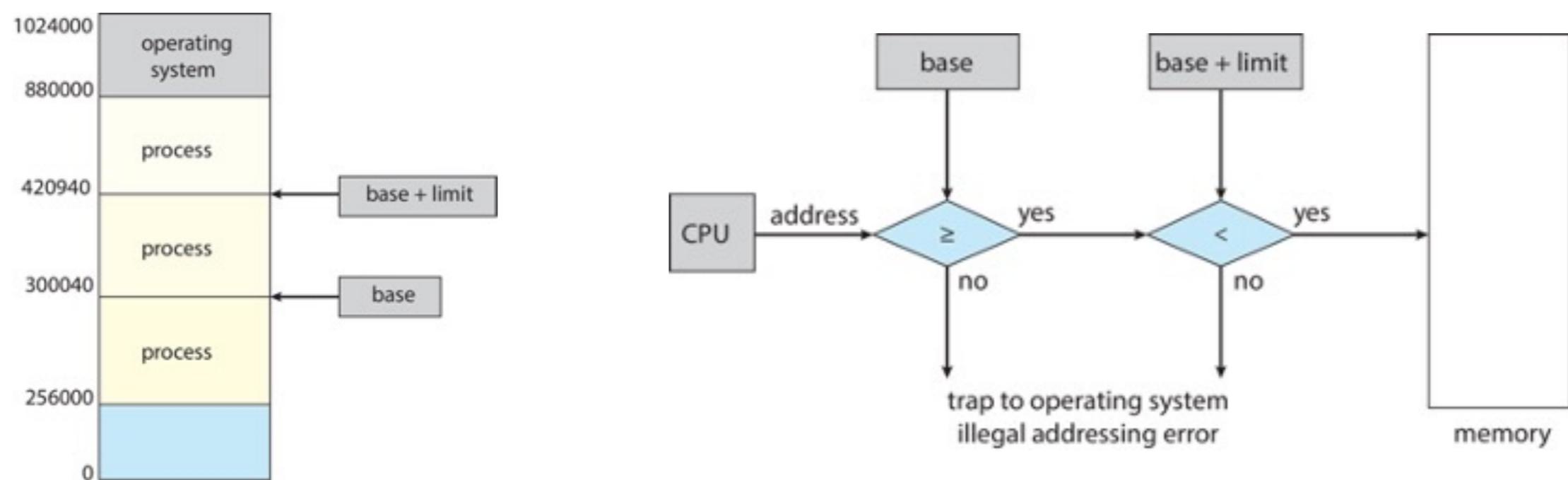


```
aaronkilik@tecmint ~ $ cat /proc/swaps
Filename           Type      Size   Used   Priority
/mnt/swapfile     file      2097148 131736 -1
aaronkilik@tecmint ~ $
```

Size	Used	Priority
2097148	131736	-1

메모리 보호 기법

- ▶ CPU는 사용자 모드에서 생성된 모든 메모리 액세스를 확인하여 해당 사용자의 기본값과 제한값 사이에 있는지 확인해야 함. → 하드웨어, 즉 MMU의 지원을 받음
- ▶ 프로세스의 논리적 주소 공간을 정의하는 Base register와 Limit register를 사용하여 접근 보호 기능 → 애를 벗어나는 주소 공간을 접근하면 에러!



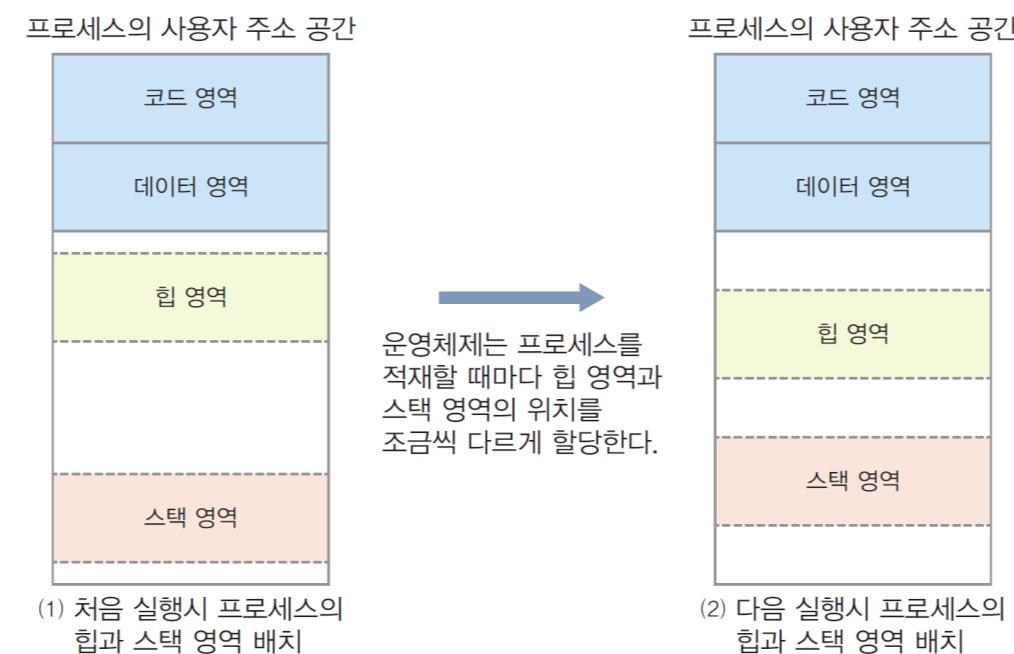
ASLR (Address Space Layout Randomization)

- ▶ 메모리 주소가 누출되는 일은 보안상 매우 위험한 일!
 - 임의의 코드 실행 → 그 코드가 system 함수라면? → 뭐든 다 할 수 있음!

▶ 주소 공간의 랜덤 배치

- 프로세스의 주소 공간 내에서 스택이나 힙, 라이브러리 영역의 랜덤 배치
 - 직접적인 메모리 참조가 힘들어짐
- 실행할 때마다 이들의 논리 주소가 바뀌게 하는 기법.

실행할 때마다 함수의 지역 변수와 동적 할당 받는 메모리의 논리 주소가 바뀜

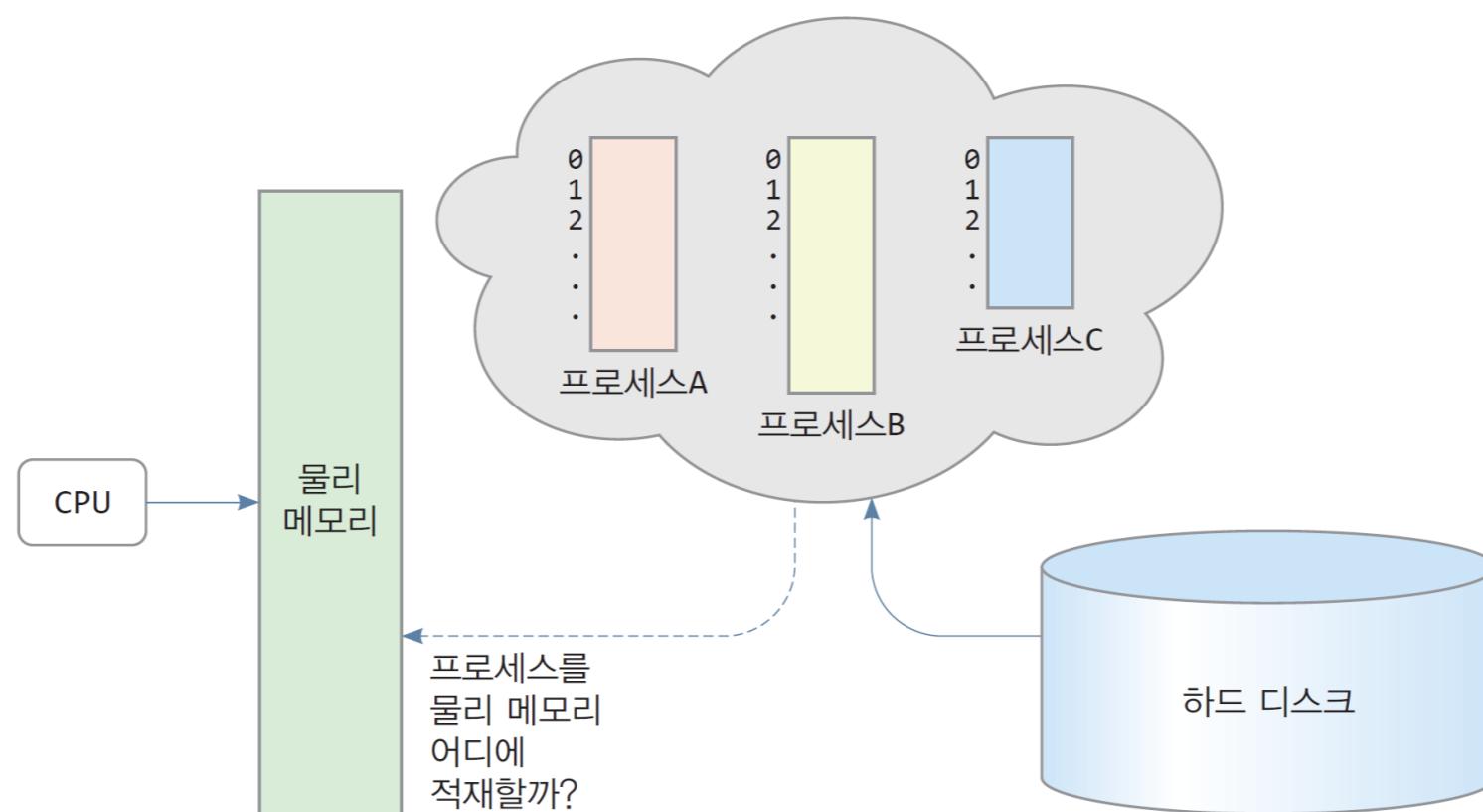


메모리 할당

그리고 단편화

Memory allocation

- ▶ 운영체제가 새 프로세스를 실행시키기거나 실행 중인 프로세스가 메모리를 필요로 할 때, 물리 메모리를 할당해줘야 함.
 - 프로세스의 실행은 할당된 물리 메모리에서 이루어짐
 - 프로세스의 코드(함수), Data, Heap, stack 등
- ▶ 올리는 건 좋은데... 어디에, 어떻게 올려야 할까?



메모리 할당 기법

▶ 연속 vs 불연속: Process partitioning

- **연속:** 프로세스가 메모리 내에 인접되어 연속되게 하나의 블록을 차지하도록 배치
- **불연속(분산):** 프로세스가 페이지나 세그먼트 단위로 나뉘어 분산 배치되는 방법



• 가변 vs 고정: Memory partitioning

- **가변:** 프로세스의 크기에 맞게 메모리가 분할
 - 메모리의 영역이 각각 다름
- **고정:** 프로세스의 크기에 상관없이 메모리가 같은 크기로 나뉨
 - 큰 프로세스가 메모리에 올라오면 여러 조각으로 나누어 배치

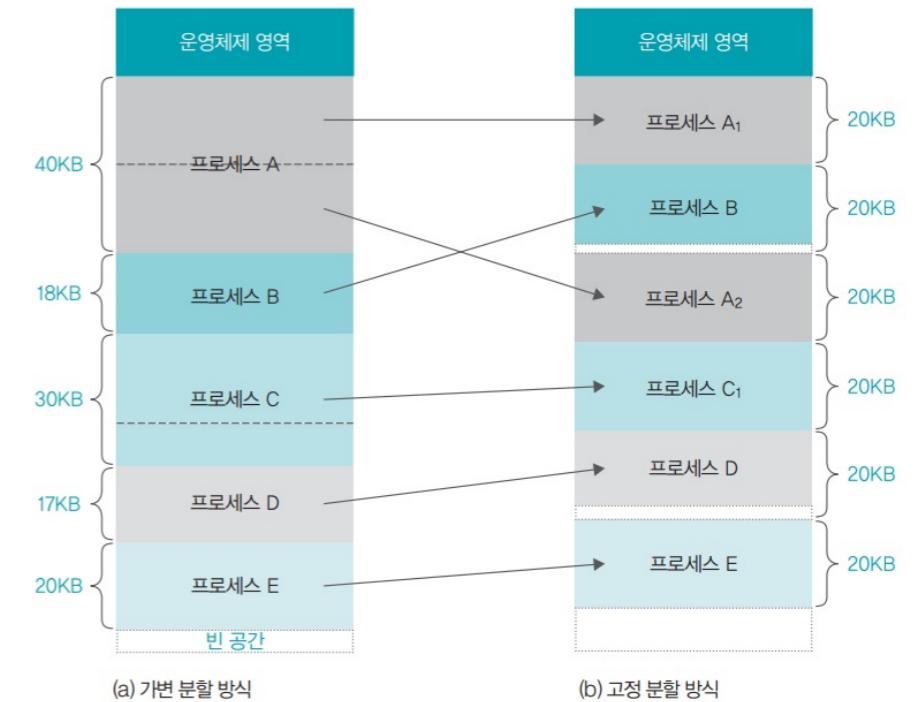
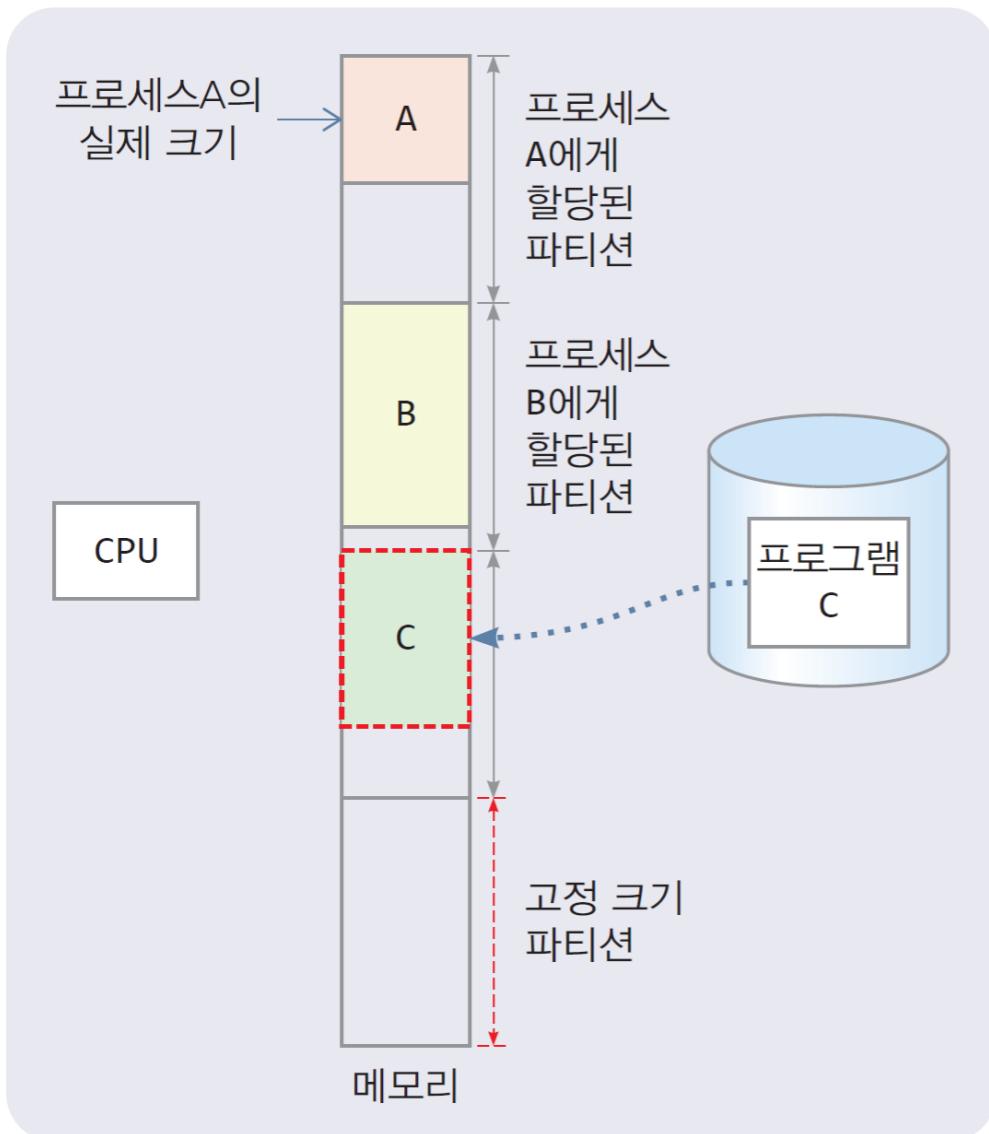


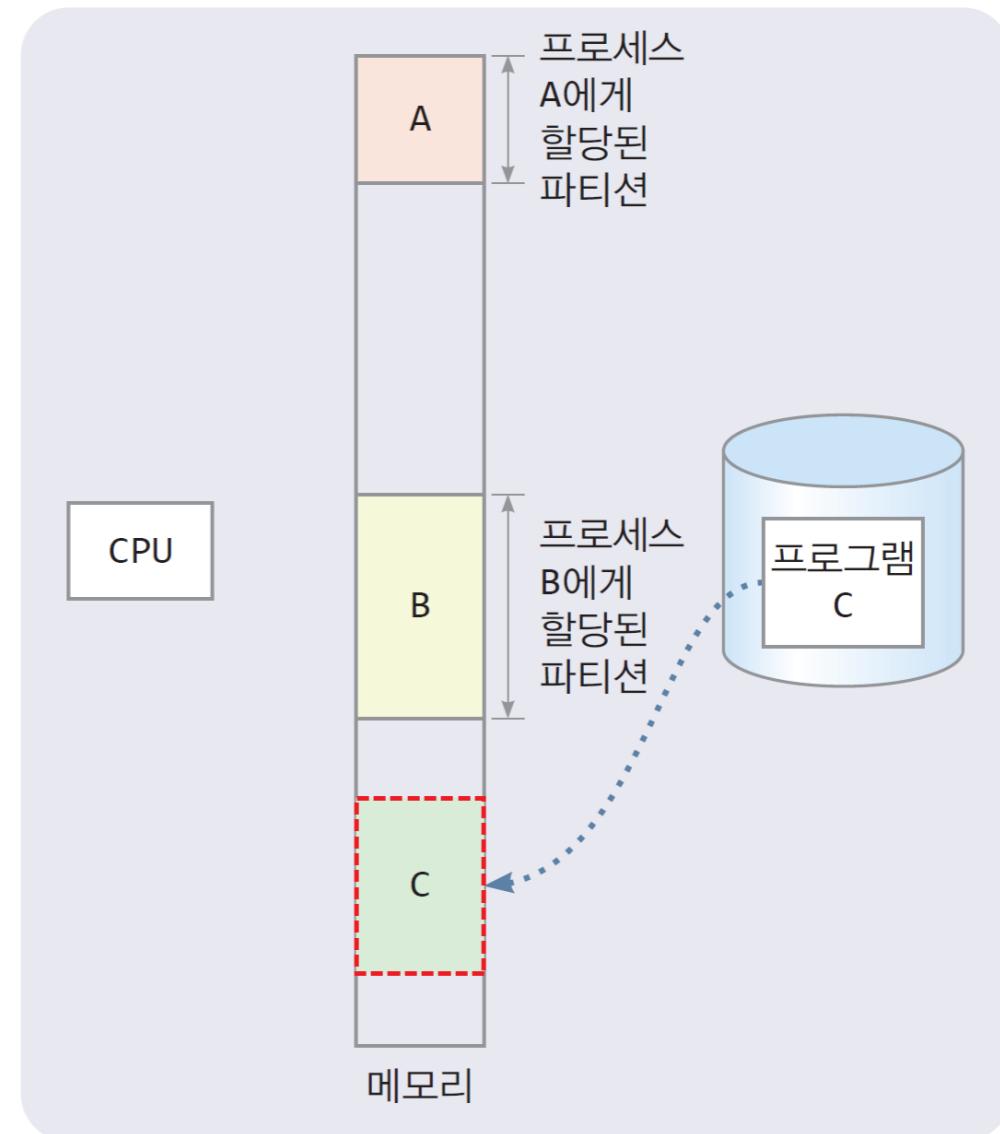
그림 7-24 메모리 분할 방식의 구현

연속 메모리 할당 (Contiguous Allocation)



메모리를 고정 크기의 파티션으로 나누고
각 프로세스를 하나의 파티션에 배치

(a) 연속 메모리 할당 – 고정 크기

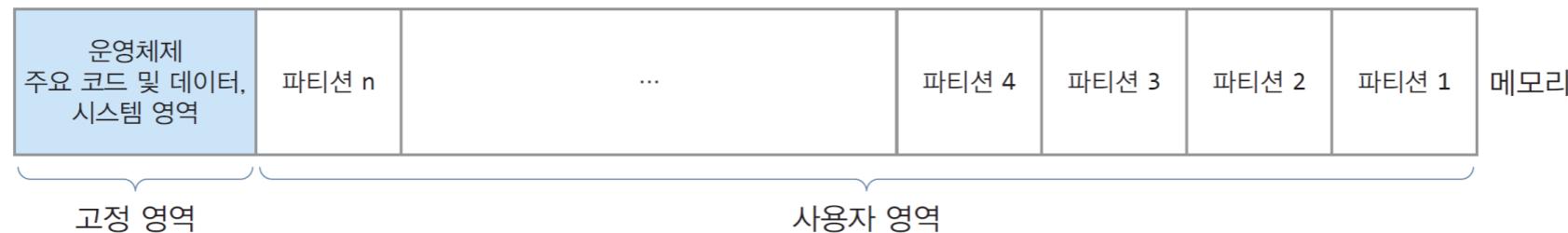


각 프로세스에게 자신의 크기만한
크기의 파티션 할당

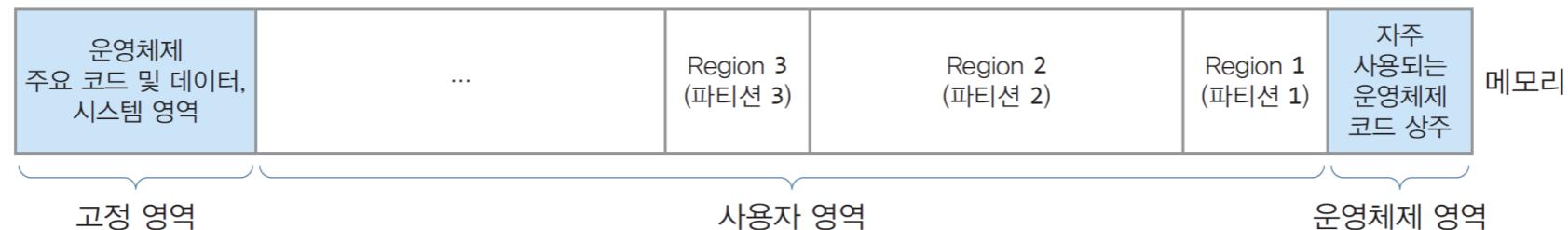
(b) 연속 메모리 할당 – 가변 크기

연속 메모리 할당

- ▶ 각 프로세스의 영역(코드와 데이터)을 연속된 메모리 공간에 배치
 - 메모리를 한 개 이상의 파티션으로 분할하고 파티션을 할당하는 기법
 - 한 프로세스는 한 파티션으로 할당
- ▶ 고정 크기(fixed size partition) 할당 ~ MFT(Multiple Programming with a Fixed Number of Tasks)
 - 메모리 전체를 고정 크기의 n개로 분할. 프로세스마다 하나씩 할당. 수용가능 프로세스의 수 n 고정



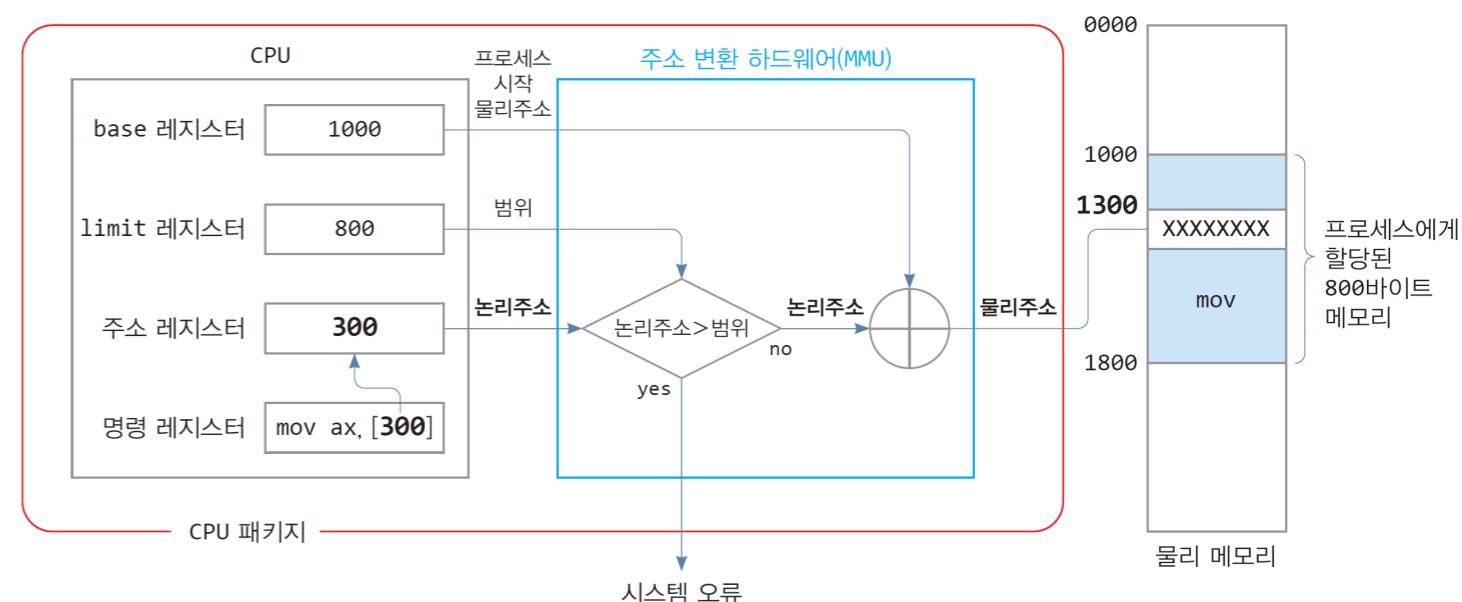
- ▶ 가변 크기(variable size partition) 할당 ~ MVT(Multiple Programming with a Variable Number of Tasks)
 - 프로세스마다 가변 크기로 연속된 메모리 할당. 수용가능 프로세스 수 가변



- ▶ 메모리가 없을 때, 프로세스는 큐에서 대기연속 메모리 할당은 초기 운영체제에서 사용

연속 메모리 할당 구현

- ▶ 하드웨어 지원
 - base 레지스터, limit 레지스터, 주소 레지스터
 - 주소 변환 하드웨어(MMU)
- ▶ 운영체제 지원
 - 모든 프로세스에 대해 프로세스별로 할당된 ‘물리메모리의 시작 주소와 크기 정보 저장’ 관리
 - 비어있는 메모리 영역 관리
 - 새 프로세스를 스케줄링하여 실행시킬 때마다, ‘물리 메모리의 시작 주소와 크기 정보’를 CPU 내부의 base 레지스터와 limit 레지스터에 적재
- ▶ 연속 메모리 할당의 장단점
 - 장점
 - 논리 주소를 물리 주소로 바꾸는 과정 단순, CPU의 메모리 액세스 속도 빠름
 - 운영체제가 관리할 정보량이 적어서 부담이 덜함
 - 단점
 - 메모리 할당의 유연성이 떨어짐.
 - 외부 단편화



연속 할당에서의 메모리 배치 기법

- ▶ 홀 선택 알고리즘/동적 메모리 할당: 운영체제는 할당 리스트(allocation list) 유지
 - 할당된 파티션에 관한 정보를 리스트로 유지 관리
 - 할당된 위치, 크기, 비어 있는지 유무
- ▶ 할당 요청이 발생하였을 때 홀 선택 전략 3가지
 - **first-fit(최초 적합)**
 - 비어 있는 파티션 중 맨 앞에 요청 크기보다 큰 파티션 선택
 - 할당 속도 빠름/단편화 발생 가능성
 - **best-fit(최적 적합)**
 - 비어 있는 파티션 중 요청을 수용하는 가장 작은 파티션 선택
 - 크기 별로 파티션이 정렬되어 있지 않으면 전부 검색
 - 가장 작은 홀 생성됨
 - **worst-fit(최악 적합)**
 - 비어 있는 파티션 중 요청을 수용하는 가장 큰 파티션 선택
 - 크기 별로 파티션이 정렬되어 있지 않으면 전부 검색
 - 가장 큰 홀 생성됨



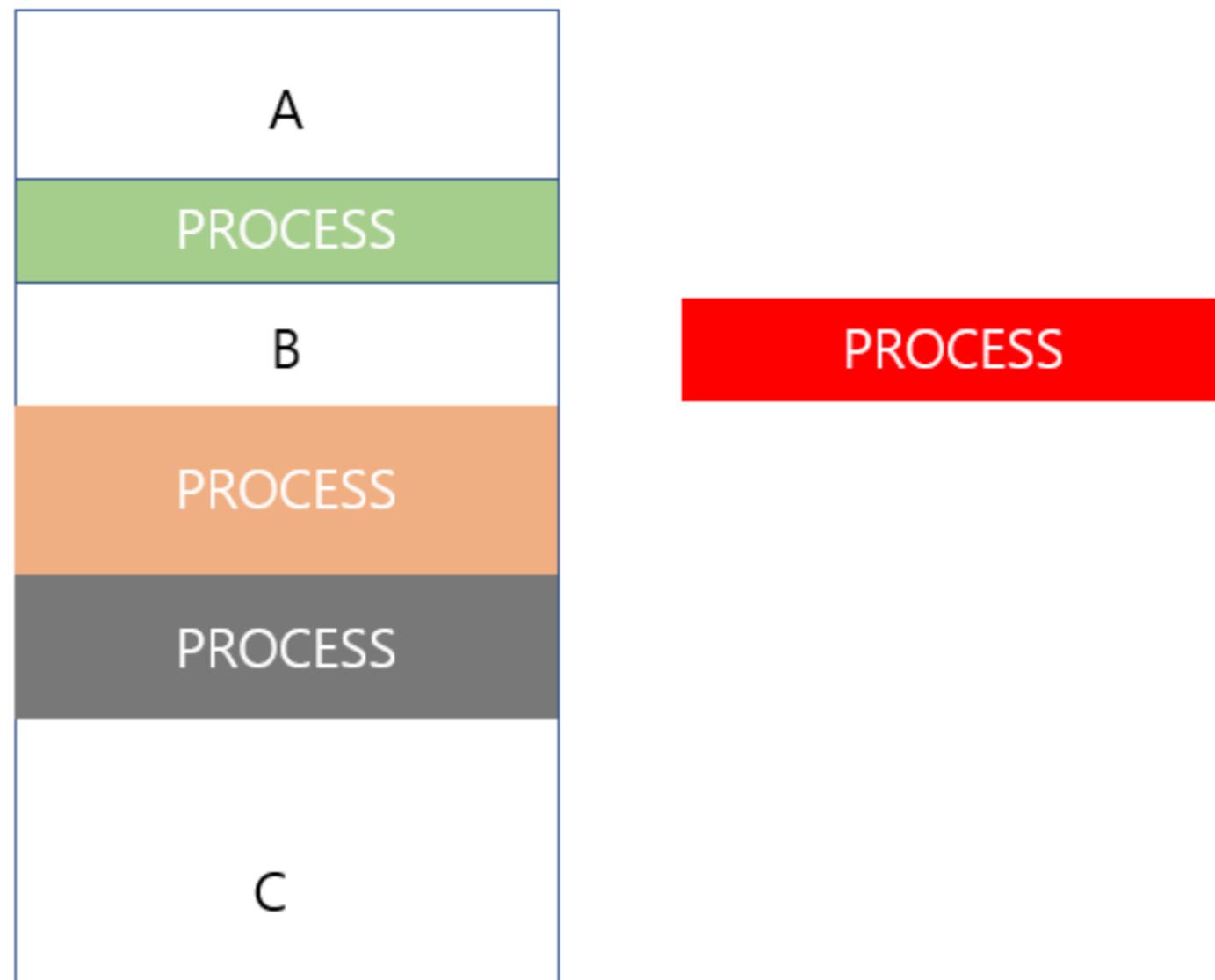
그림 7-30 메모리 배치 방식의 비교

Fit!

- ▶ First-fit

- ▶ Best-fit

- ▶ Worst-fit



뭐가 젤 좋을까?

- ▶ Simulations have shown that both first fit and best fit are better than worst fit in terms of decreasing time and storage utilization.
 - 시뮬레이션 해보면, first-fit과 best-fit이 속도나 메모리 사용률 측면에서 worst 보다 낫더라!
- ▶ first-fit vs best-fit
 - Case by case!
 - 단, 알고리즘적으로는 first-fit이 유리하다!
 - 뭐 근데 보통 그렇다더라 하는거지, 대개 큰 차이는 없더라...!
- ▶ 물론 상황마다 얼마든지 달라질 수 있음!

연속 할당에서 입출력이 반복되다보면...

- ▶ 메모리에 프로세스의 입출력이 반복되다보면... 단편화가 발생합니다!
 - 프로세스 A, B, C, D, E를 순서대로 배치했을 때 프로세스 B와 D가 종료되면 18KB와 17KB의 빈 공간이 생김
- ▶ 이후 18KB보다 큰 프로세스가 들어오면 적당한 공간이 없어 메모리를 배정 못 함
 - 총 메모리 잔여 공간? $18+17 = 35\text{KB}$.
 - 하지만, 20KB의 프로세스는 수용하지 못한다...!
- ▶ 단편화: 조각난 공간들...
 - 합치면 분명히 용량이 있지만, 조각나서 못쓰는 공간들.
 - 우측 예제의 것은 '[외부 단편화](#)'라고 부릅니다.
 - ↔ 당연히 '내부 단편화'도 있음.
 - (뒷장에 설명!)

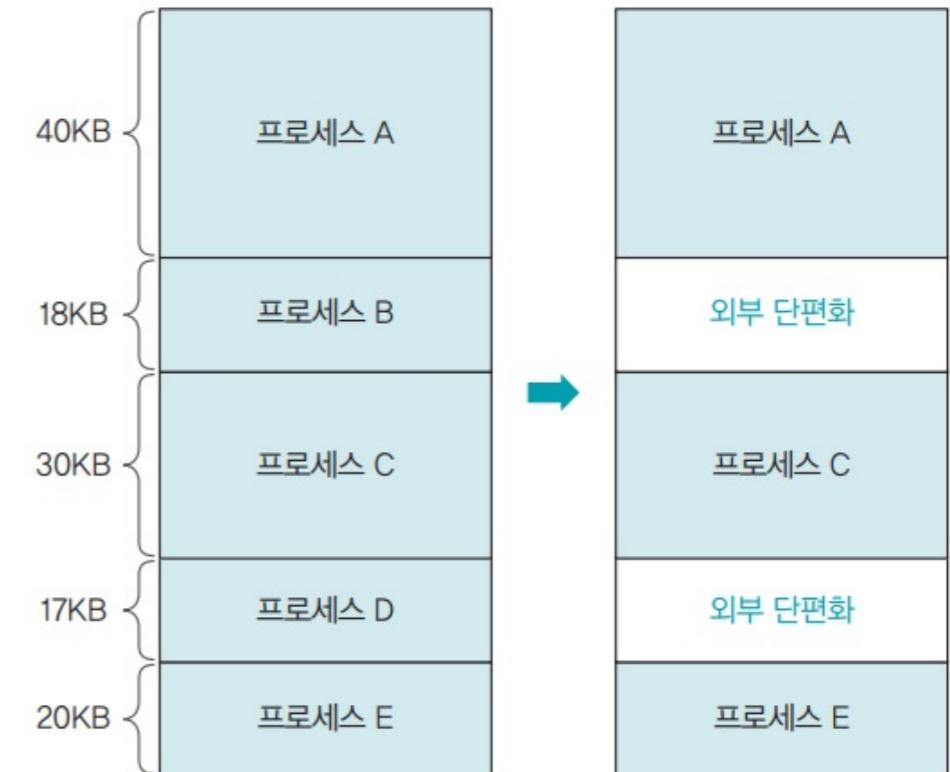
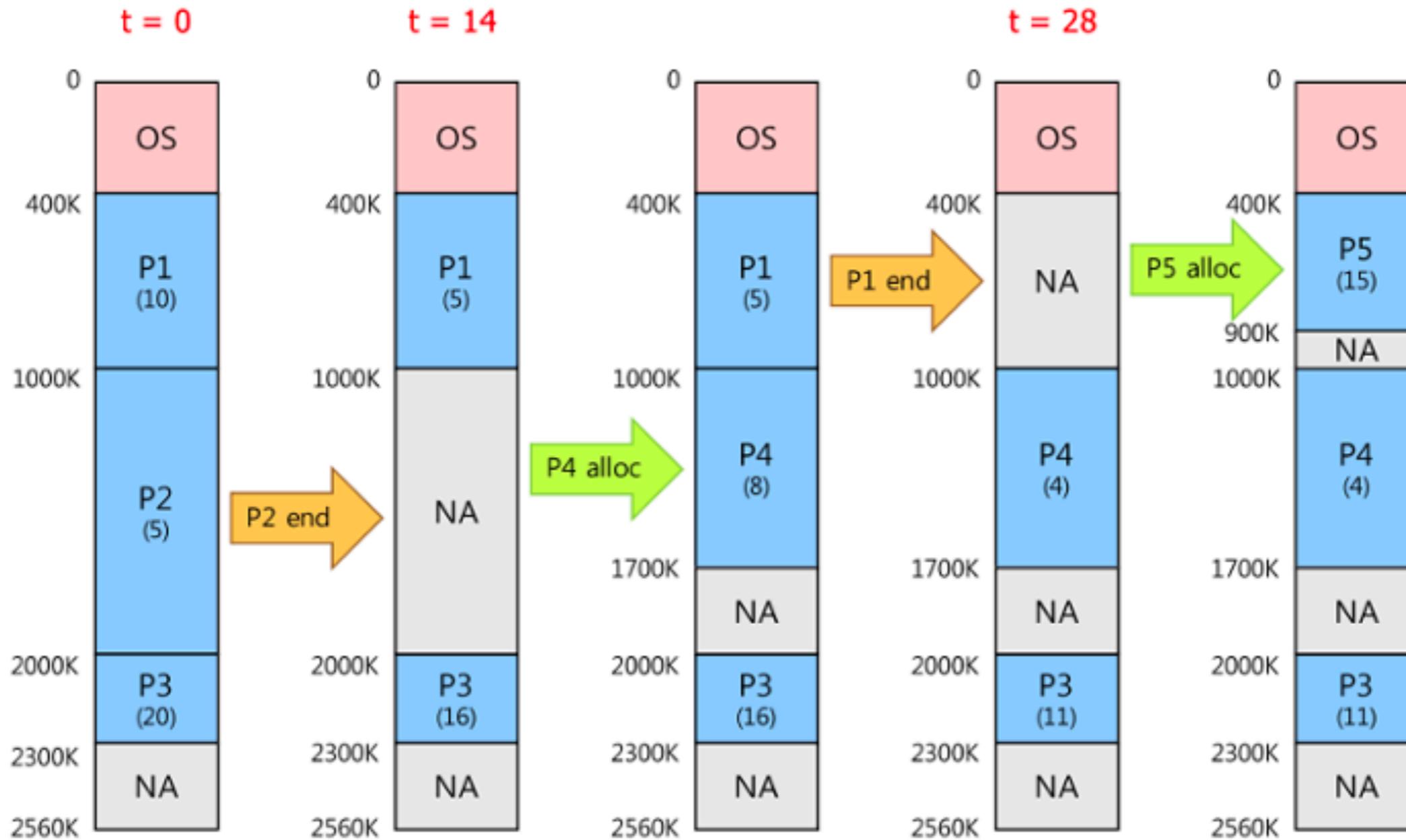


그림 7-28 가변 분할 방식과 외부 단편화

다시 봅시다.



단편화 Fragmentation

- ▶ 프로세스에게 할당할 수 없는 조각 메모리 (Hole)들이 생기는 현상,
- ▶ **외부 단편화(external fragmentation):** 할당된 **메모리들 사이**에 사용할 수 없는 훈이 생기는 현상
 - 가변 크기의 파티션이 생기고 반환되면서 여러 개의 작은 훈 생성
 - 훈이 프로세스의 크기(요구되는 메모리 량)보다 작으면 할당할 수 없음
 - MVT(Multiple Programming with a Variable Number of Tasks)의 경우



- ▶ **내부 단편화(internal fragmentation):** 할당된 **메모리 내부**에 사용할 수 없는 훈이 생기는 현상
 - 파티션보다 작은 프로세스(요구되는 메모리)를 할당하는 경우 발생
 - MFT(Multiple Programming with a Fixed Number of Tasks)의 경우



계산 해보자

- ▶ 각 파티션에 해당되는 작업들을 배치하고자 할 때...
 - 외부 단편화
 - 내부 단편화

분할영역	분할크기	작업크기
1	50	60
2	150	160
3	200	100
4	250	150

단편화 해결방법

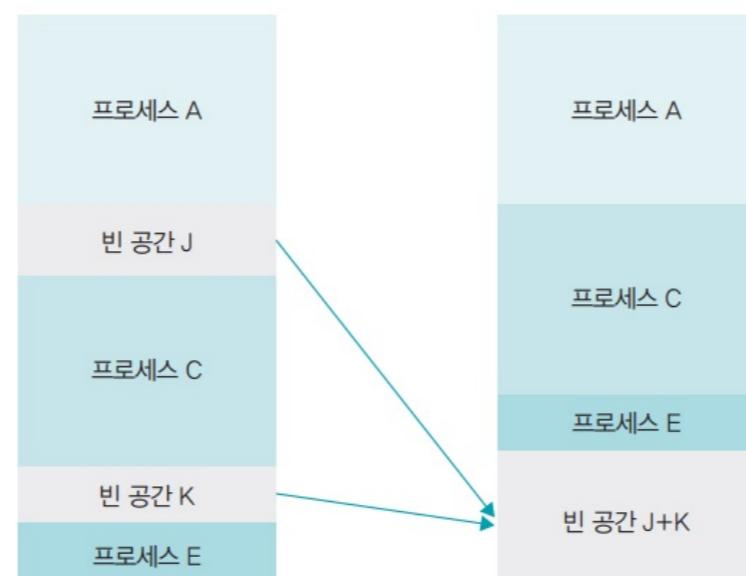
▶ 조각 모음 (De-fragmentation) / Compaction(압축)

- 이미 배치된 프로세스를 옆으로 옮겨 빈 공간들을 하나의 큰 덩어리로 만드는 작업

▶ 동작

- 조각 모음을 하기 위해 이동할 프로세스의 동작을 멈춤
- 프로세스를 적당한 위치로 이동
(프로세스가 원래의 위치에서 이동하기 때문에 프로세스의 상대 주소값을 바꿈)
- 작업을 다 마친 후 프로세스 다시 시작

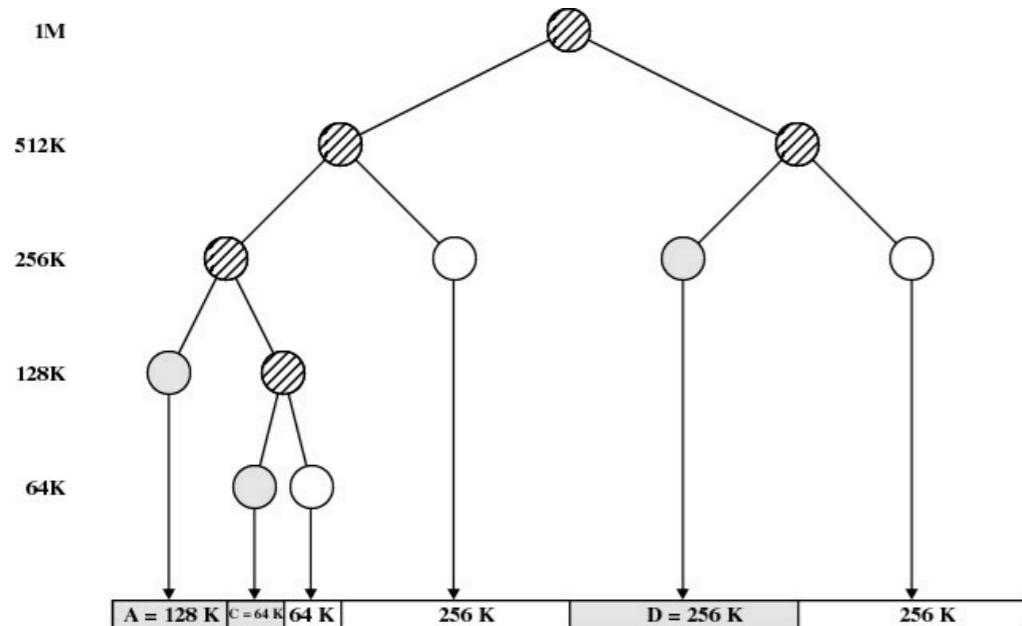
▶ 굉장히 비싼 작업...!



Buddy system

▶ 메모리 할당의 Binary search

- 1) 메인 메모리의 크기는 2^N
- 2) 사용할 수 있는 가장 큰 메모리부터 시작해서 Binary로 절반씩 쪼개나가면서 아래 조건을 만족시키는 공간을 찾는다.
 - 3) 프로세스의 크기가 K일 때, $2^{U-1} < K \leq 2^U$ 를 만족시키는 U를 찾고, 2^U 크기의 공간에 할당.
 - 예를 들어, 프로세스 크기가 100이면, $64 < 100 < 128$ 이기 때문에 128 사이즈의 메모리에 할당
- 4) 프로세스가 종료되고, 만약 같은 Parent를 갖는 Buddy 공간이 비어있다면 Merge.

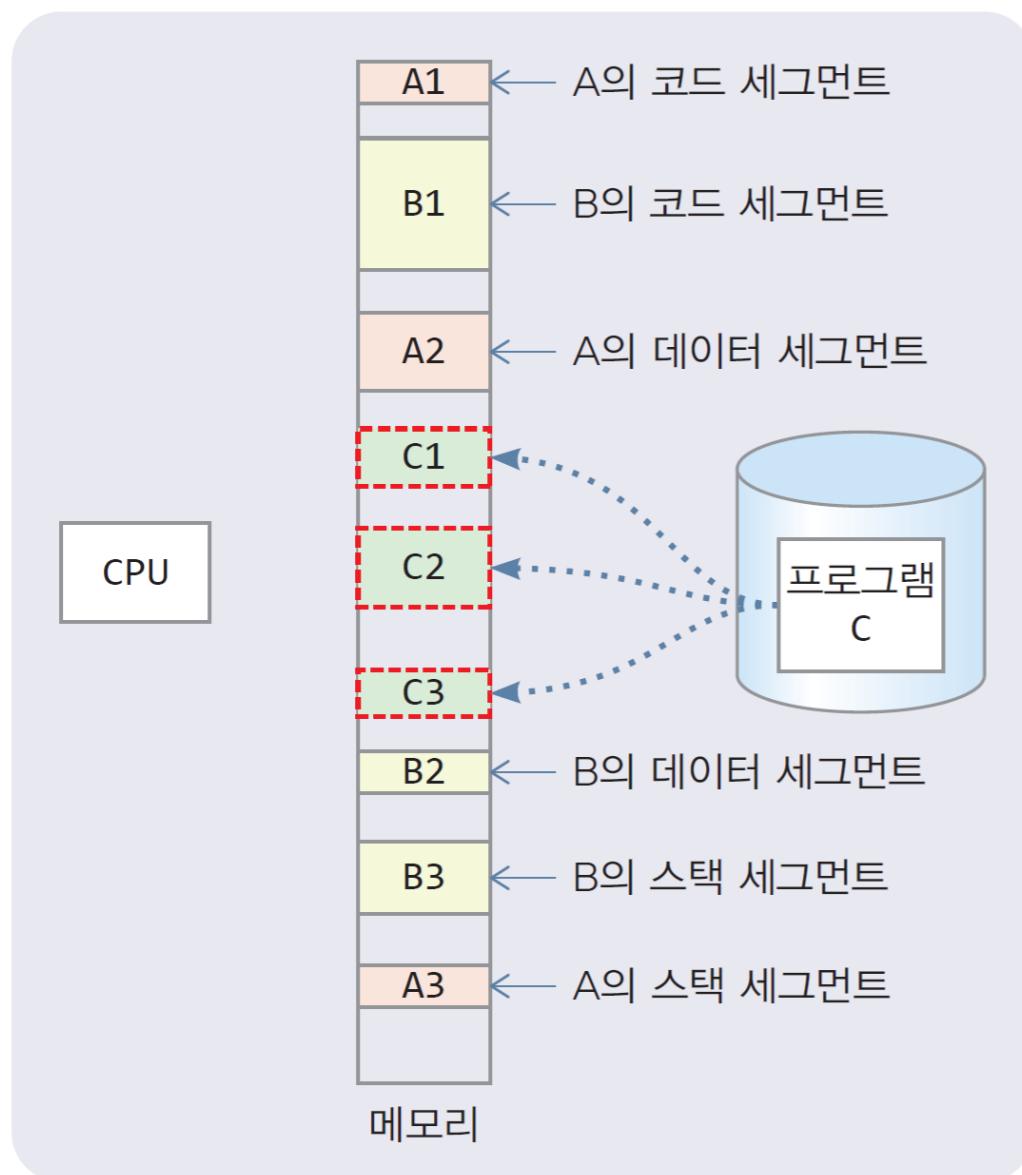


1 Mbyte block	1 M
Request 100 K	A = 128 K 128 K 256 K 512 K
Request 240 K	A = 128 K 128 K B = 256 K 512 K
Request 64 K	A = 128 K C = 64 K 64 K B = 256 K 512 K
Request 256 K	A = 128 K C = 64 K 64 K B = 256 K D = 256 K 256 K
Release B	A = 128 K C = 64 K 64 K 256 K D = 256 K 256 K
Release A	128 K C = 64 K 64 K 256 K D = 256 K 256 K
Request 75 K	E = 128 K C = 64 K 64 K 256 K D = 256 K 256 K
Release C	E = 128 K 128 K 256 K D = 256 K 256 K
Release E	512 K D = 256 K 256 K
Release D	1 M

버디 시스템 특징

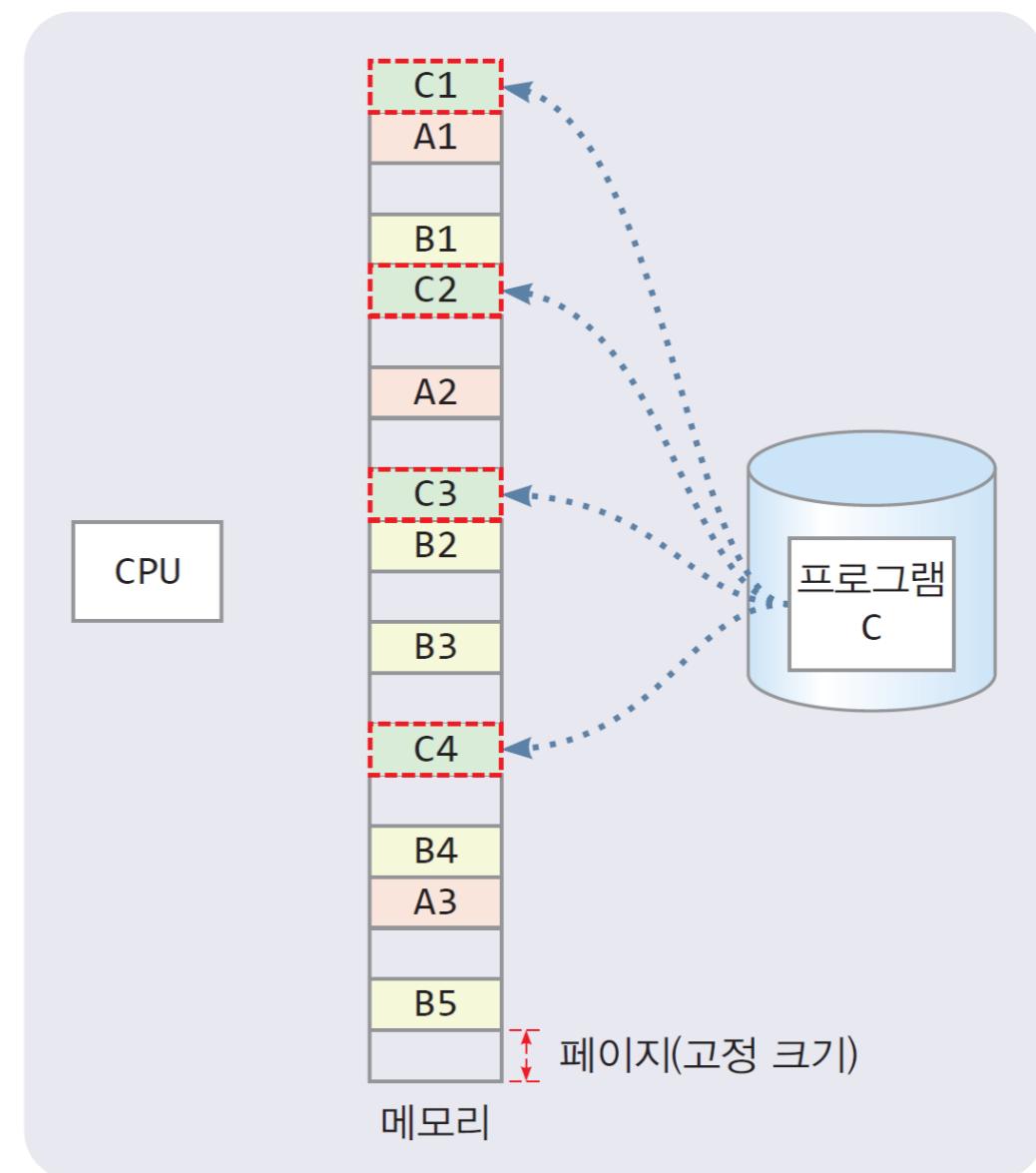
- ▶ 가변 분할 방식처럼 메모리가 프로세스 크기대로 나뉨
- ▶ 고정 분할 방식처럼 하나의 구역에 다른 프로세스가 들어갈 수 없고, 메모리의 한 구역 내부에 조각이 생겨 **내부 단편화** 발생
- ▶ 비슷한 크기의 조각이 서로 모여 작은 조각을 통합하여 큰 조각을 만들기 쉬움

분할 메모리 할당 (Noncontiguous Allocation)



프로세스를 가변 크기의 세그먼트들로 분할 할당

(c) 분할 할당 – 세그먼테이션(segmentation)



프로세스를 고정 크기의 페이지들로 분할 할당

(d) 분할 할당 – 페이징(paging)

분할 메모리(불연속 메모리) 할당

- ▶ 분할된 크기는 20KB이므로 40KB인 프로세스 A는 프로세스 A1과 A2로 나뉘어 할당
- ▶ 30KB인 프로세스 C는 프로세스 C1과 C2로 나뉘는데, 메모리에 남은 공간이 없으므로 프로세스 C2는 스왑 영역으로 옮겨짐

- ▶ 정확히 우측 그림은
 - 불연속이면서 고정 분할이면서
 - 자세한건 다음시간, 페이징을 배울 때!

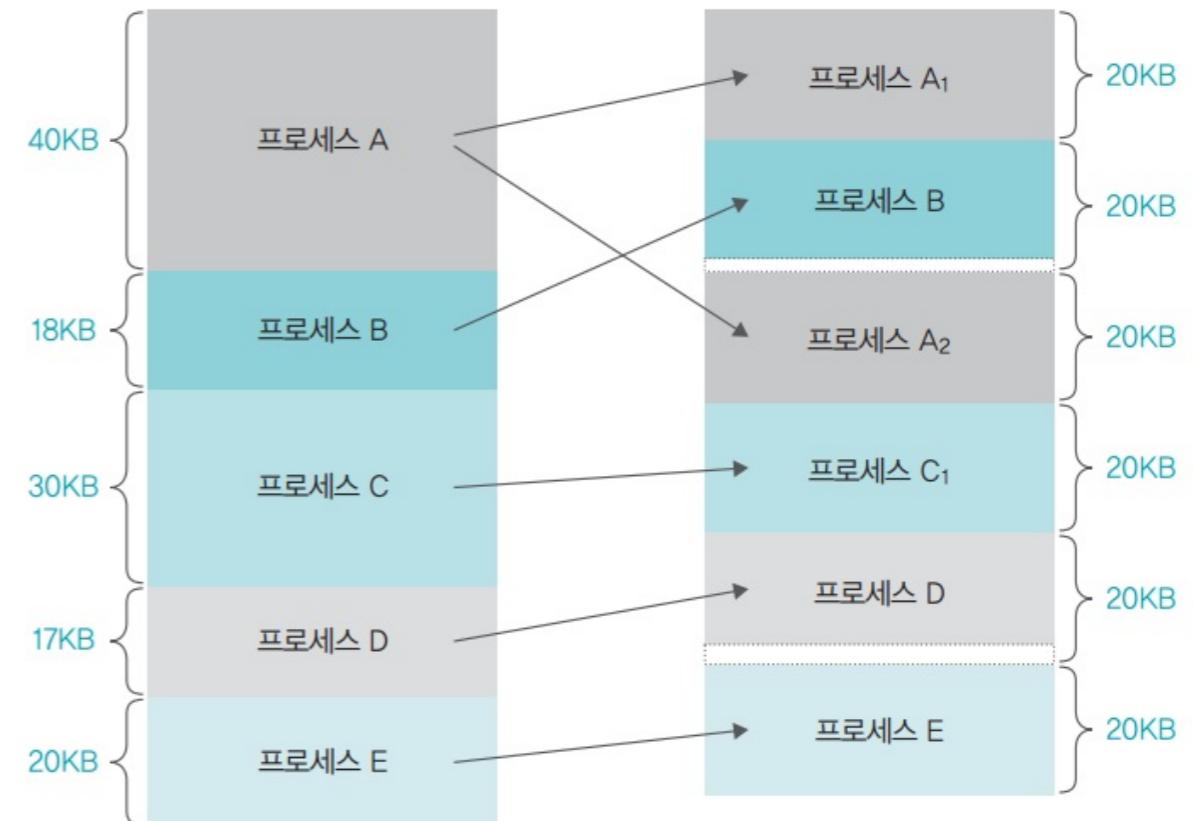


그림 7-34 고정 분할 방식의 프로세스 배치

세그먼테이션(segmentation) 개요

▶ 세그먼트(segment)

- 세그먼트는 논리적 단위 – 개발자의 관점에서 보는 프로그램의 논리적 구성 단위
- 프로그램을 구성하는 일반적인 세그먼트 종류
 - 코드 세그먼트, 데이터 세그먼트, 스택 세그먼트, 힙 세그먼트
- 세그먼트마다 크기 다름

▶ 세그먼테이션 기법

- 프로세스를 논리 세그먼트 크기로 나누고, 각 논리 세그먼트를 한 덩어리의 물리 메모리에 할당하고 관리

▶ 프로세스의 주소 공간

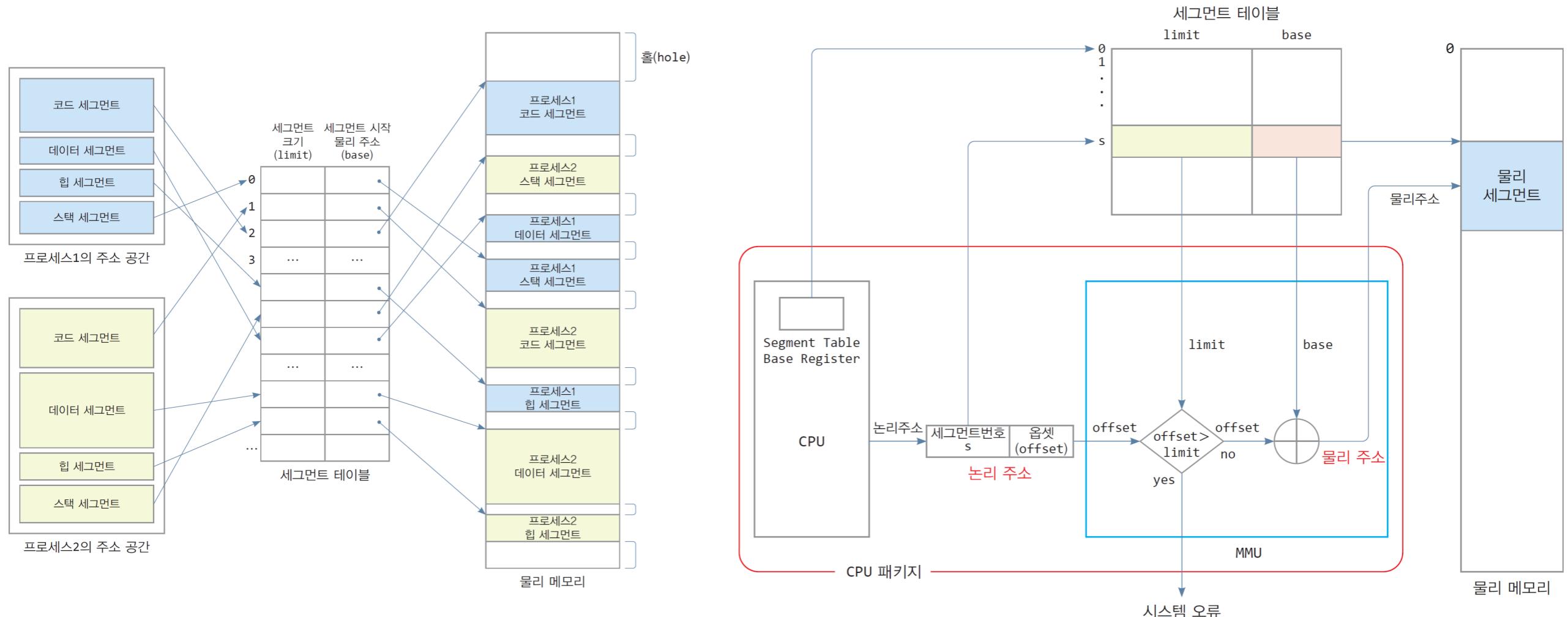
- 프로세스의 주소 공간을 여러 개의 논리 세그먼트들로 나누고
- 각 논리 세그먼트를 물리 세그먼트에 맵핑
- 프로세스를 논리 세그먼트로 나누는 과정은 컴파일러, 링커, 로더, 운영체제에 의해 이루어짐

▶ 논리 세그먼트와 물리 세그먼트의 맵핑

- 시스템 전체 세그먼트 맵핑 테이블을 두고 논리 주소를 물리 주소로 변환

▶ 외부 단편화 발생

논리 세그먼트와 물리 세그먼트 매핑



Summary

- ▶ 메모리 관리 → 컴퓨터를 '잘' 쓰기 위해서 매우 중요함!
 - 적재 정책, 배치 정책, 재배치 정책
- ▶ 메모리 주소
 - 절대 주소(물리주소), 상대주소(가상주소) → MMU를 통해 변환!
 - 메모리 Overlay와 Swap
- ▶ 메모리 할당
 - 연속 vs 불연속, 가변 vs 고정
 - 메모리 배치 기법: First, best, worst
 - 단편화! → 압축
 - buddy system
- ▶ Segmentation,
 - 프로세스를 각 세그먼트 단위로 나눠서 적재.
 - 오늘날 메모리는 대부분 Paging! 그리고 가상메모리와 관련된 이야기들...!