Chapter

08

메모리 관리

1. 메모리 계층 구조와 메모리 관리 핵심
2. 메모리 주소
3. 물리 메모리 관리
4. 연속 메모리 할당
5. 세그먼테이션 메모리 관리

강의 목표

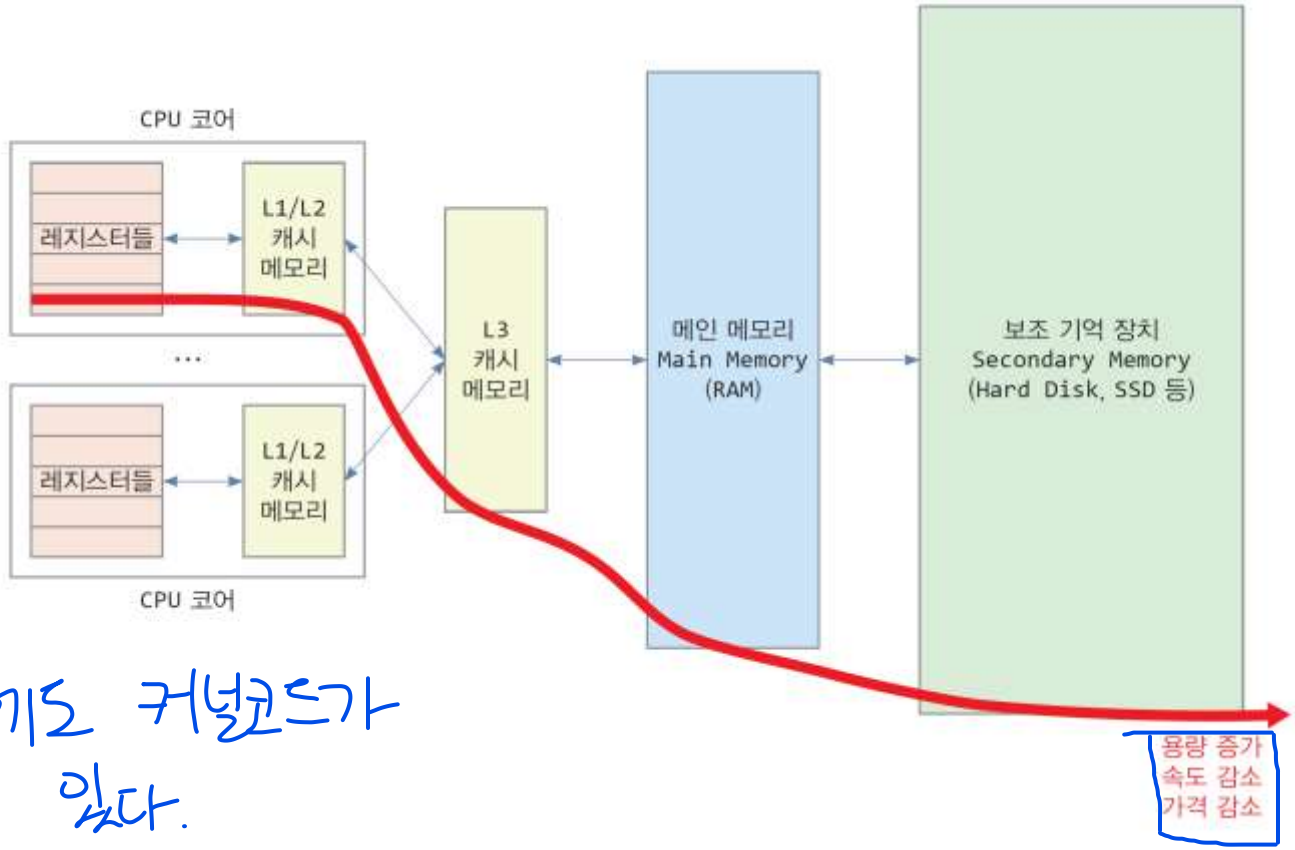
1. 컴퓨터 시스템에 존재하는 다양한 저장소들로 구성되는 메모리 계층 구조를 이해하고 필요성을 안다.
2. 메모리에 대한 물리 주소와 논리 주소를 이해하고 프로그램 실행 중에 논리 주소가 물리 주소로 변환됨을 이해한다.
3. 프로세스의 실행에 필요한 메모리 할당 정책에 대해 이해한다.
   - 연속 메모리 할당
   - 분할 메모리 할당
4. 모든 메모리 할당에는 사용할 수 없는 조각 메모리(단편화)가 발생하는데, 단편화에 대해 이해한다.
5. 홀 선택(동적 메모리 할당) 알고리즘을 이해한다.
   - first-fit, best-fit, worst-fit
6. 메모리 관리 기법 중 세그먼테이션을 구체적으로 이해한다.

3

1. 메모리 계층 구조와 메모리 관리 핵심

메모리 계층 구조

- 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



캐시메모리에도 커널코드가 올라올 수 있다.

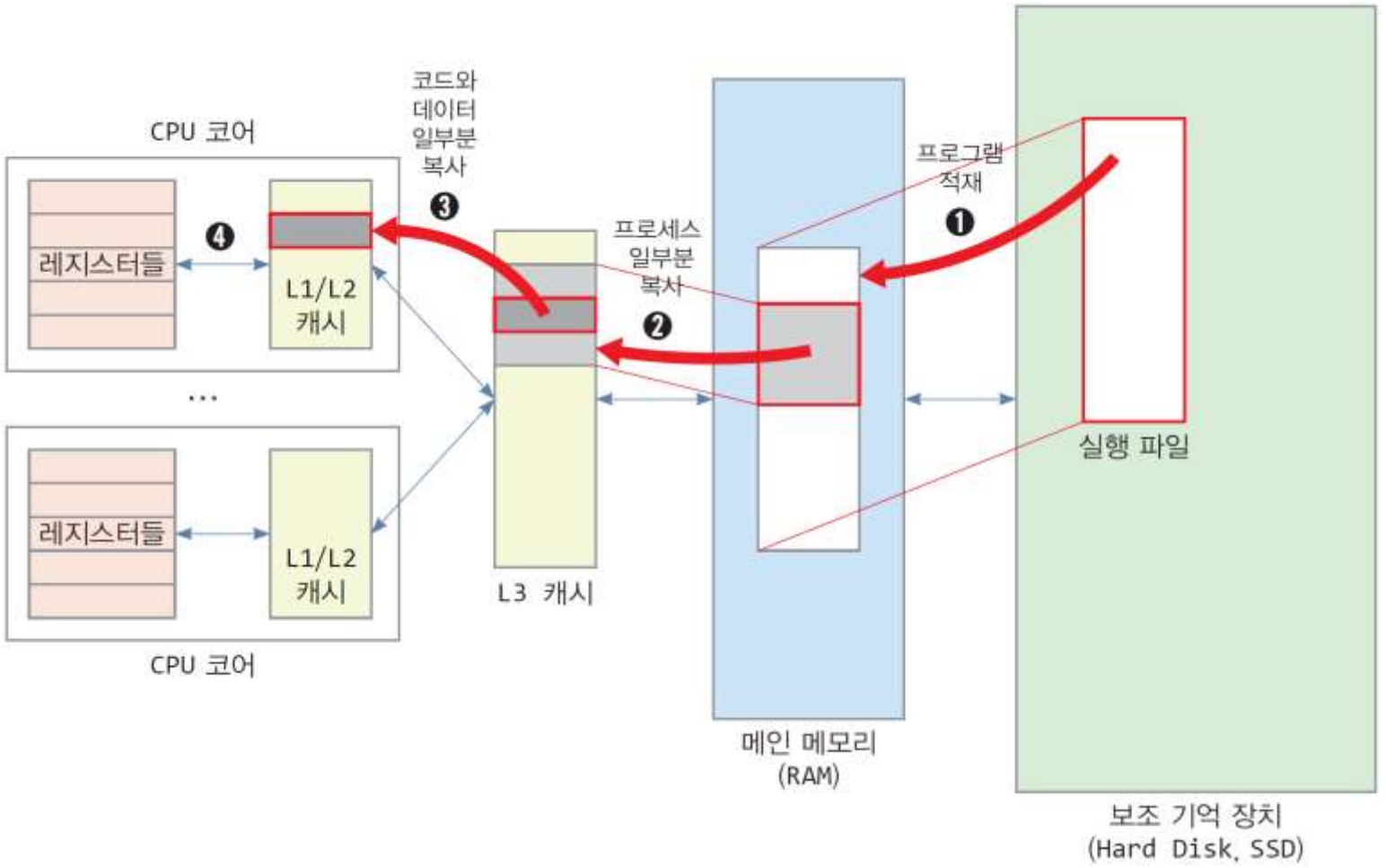
메모리 계층 구조의 특성

	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 F/F (Static RAM)	SRAM F/F (Static RAM)	DRAM 계속 refresh (Dynamic RAM)	마그네틱 필드나 플래시 메모리
속도	〈1ns	〈5ns	〈5ns	〈50ns	〈20ms
가격		고가	고가	보통	저가
휘발성	휘발성	휘발성	휘발성	휘발성	비휘발성

메모리 계층화의 목적

- 계층화의 역사적 과정
  - CPU 성능 향상 -> 더 빠른 메모리 요구 -> 작지만 빠른 off-chip 캐시 등장 -> 더 빠른 액세스를 위해 on-chip 캐시 -> 멀티 코어의 성능에 적합한 L1, L2, L3 캐시
  - 컴퓨터 성능 향상 -> 처리할 데이터도 대형화 -> 저장 장치(하드 디스크)의 대형화 -> 빠른 저장 장치 요구 -> SSD 등 등장
  - 메모리 계층화는 성능과 비용의 절충
    - 빠른 메모리일수록 고가이므로 작은 용량 사용
- 메모리 계층화의 목적
  - CPU의 메모리 액세스 시간을 줄이기 위함
    - 빠른 프로그램 실행을 위해

메모리 계층에서 코드와 데이터 이동



메모리 계층화 성공 이유

- 질문) 작은 캐시에 당장 실행할 프로그램 코드와 데이터를 일부분만 두는데도 효과적일까?

- 답) 메모리 계층화 성공 이유?
  - 참조의 지역성 때문
    - 코드나 데이터, 자원 등이 아주 짧은 시간 내에 다시 사용되는 특성 -> CPU는 작은 캐시 메모리에 적재된 코드와 데이터로 한동안 실행
    - 캐시를 채우는 시간의 손해보다 빠른 캐시를 이용하는 이득이 큼

메모리 관리

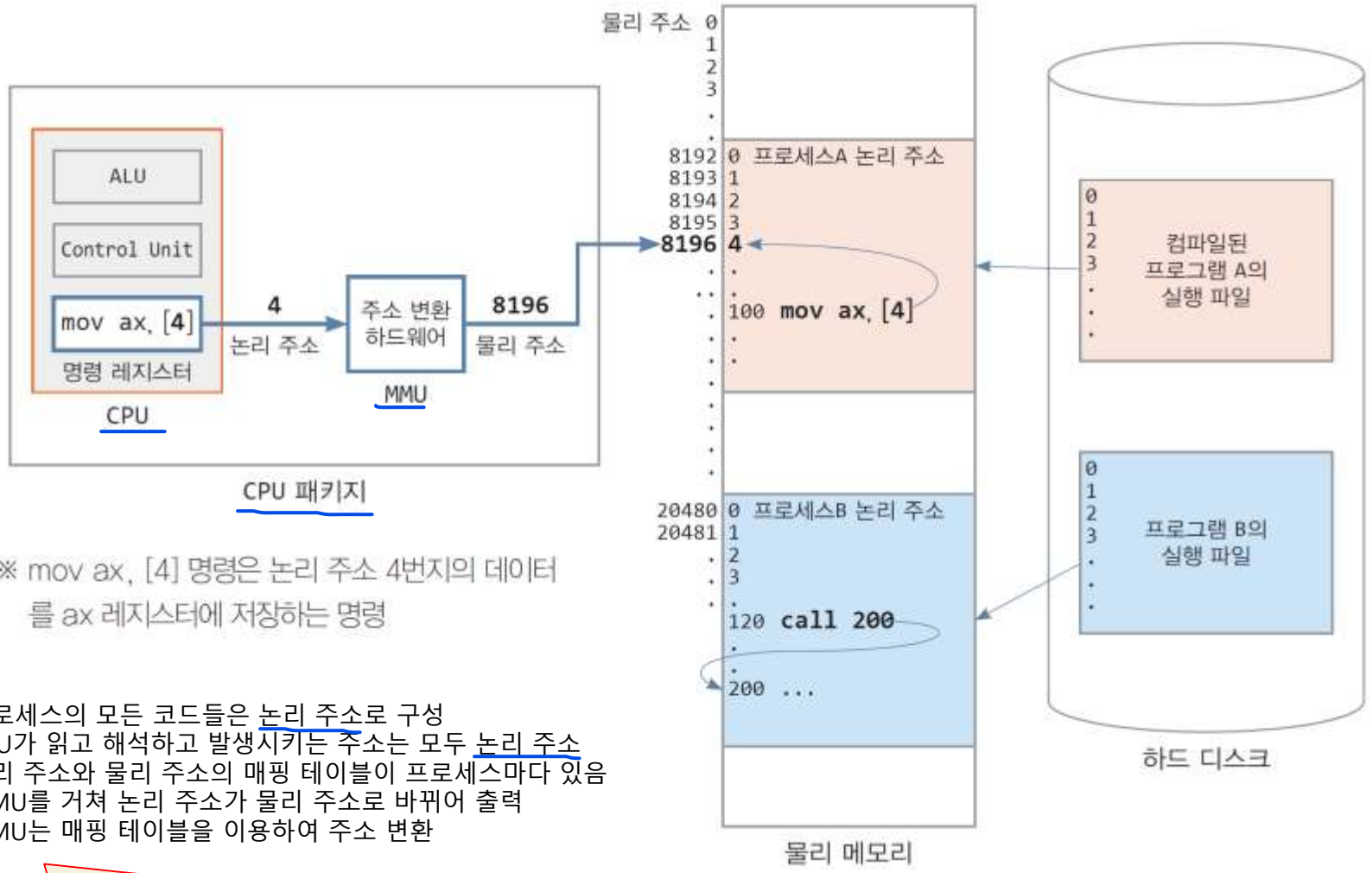
- 메모리의 역할
  - 메모리는 실행하고자 하는 프로그램 코드와 데이터 적재
  - CPU는 메모리에 적재된 코드와 데이터만 처리
- 운영체제에 의해 메모리 관리가 필요한 이유
  - 메모리는 **공유 자원이기 때문** – 8,9장에서 다룸
    - 여러 프로세스 사이에 메모리 공유
    - 각 프로세스에게 물리 메모리 할당
  - 메모리 **보호되어야 하기 때문** – 8,9장에서 다룸
    - 프로세스의 독립된 메모리 공간 보장
      - 다른 프로세스로부터 보호
    - 사용자 코드로부터 커널 공간 보호
  - 메모리 **용량 한계 극복할 필요** – 10장에서 다룸
    - 설치된 물리 메모리보다 큰 프로세스 지원 필요
    - 여러 프로세스의 메모리 합이 설치된 물리 메모리보다 큰 경우 필요
  - 메모리 **효율성 증대를 위해** – 10장에서 다룸
    - 가능하면 많은 개수의 프로세스를 실행시키기 위해
      - 프로세스당 최소한의 메모리 할당

2. 메모리 주소

물리 주소와 논리 주소

- 메모리는 오직 주소로만 접근
- 주소의 종류
  - **물리 주소**(physical address)
    - 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)
    - 개발자나 프로세스가, 프로세스 내에서 사용하는 주소, 코드나 변수 등에 대한 주소
    - 0에서 시작하여 연속되는 주소 체계, 프로세스 내에서 매겨진 상대 주소
      - 프로그램에서 변수 n의 주소가 100번지라면, 논리 주소가 100이고, 물리 주소를 알 수 없음
    - 컴파일러와 링커에 의해 매겨진 주소
      - 실행 파일에 내에 만들어진 목적 코드와 데이터의 주소들은 논리 주소로 되어 있음
    - CPU 내에서 프로세스를 실행하는 동안 다루는 모든 주소는 논리 주소
    - 사용자나 프로세스는 결코 물리 주소를 알 수 없음
- MMU(Memory Management Unit)
  - 논리 주소를 물리 주소로 바꾸는 하드웨어 장치
    - CPU가 발생시킨 논리 주소는 MMU에 의해 물리 주소로 바뀌어 물리 메모리에 도달
  - 오늘날 MMU는 CPU 패키지에 내장
    - 인텔이나 AMD의 x86 CPU는 80286부터 MMU를 내장
    - MMU 덕분으로 여러 프로세스가 하나의 물리 메모리에서 실행되도록 됨

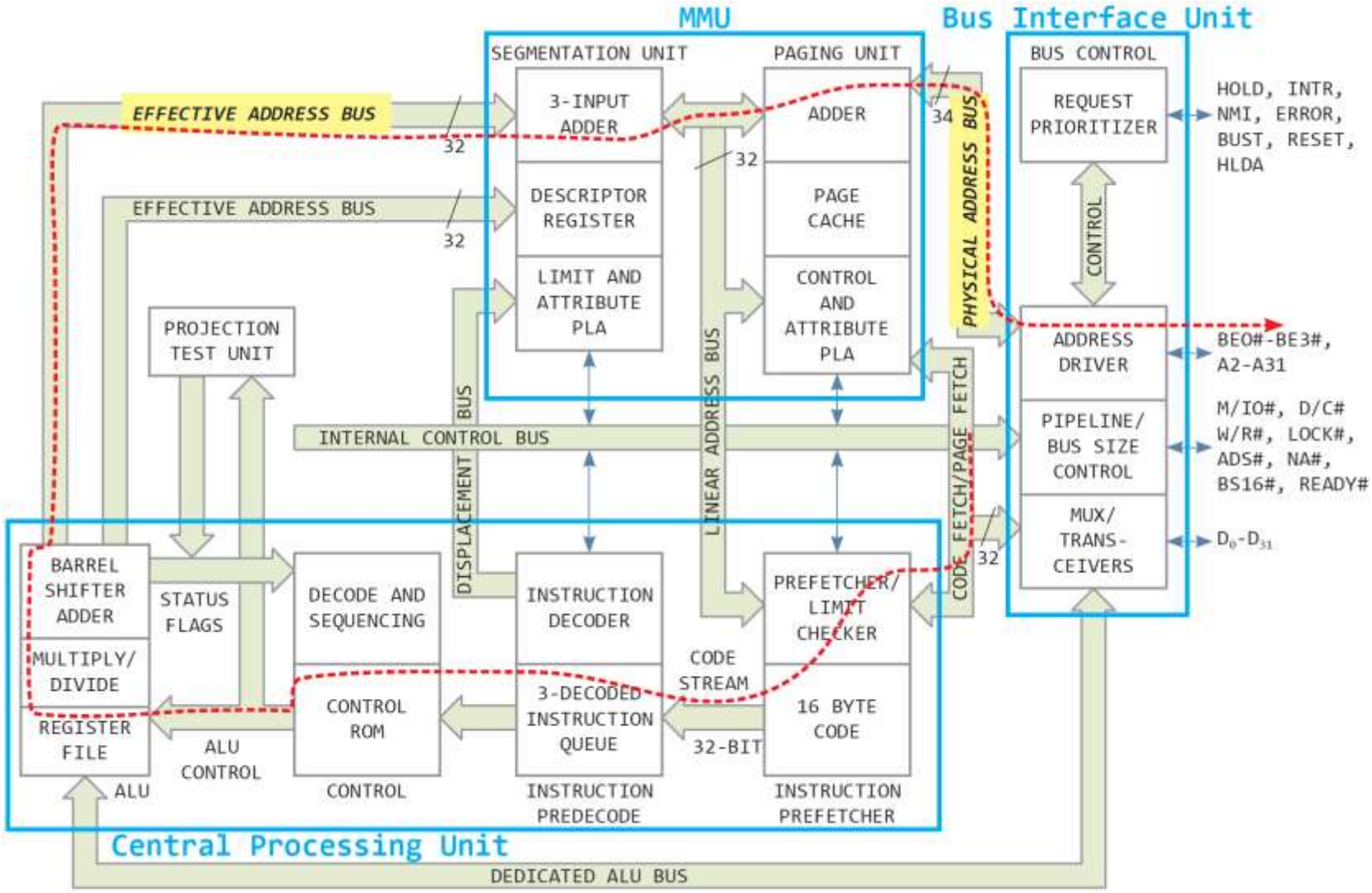
논리 주소와 물리 주소, MMU에 의한 주소 변환



- 프로세스의 모든 코드들은 논리 주소로 구성
- CPU가 읽고 해석하고 발생시키는 주소는 모두 논리 주소
- 논리 주소와 물리 주소의 매핑 테이블이 프로세스마다 있음
- MMU를 거쳐 논리 주소가 물리 주소로 바뀌어 출력
- MMU는 매핑 테이블을 이용하여 주소 변환

프로세스 A의 코드 실행 중 CPU 안의 mov ax, [4] 명령에 담긴 4 번지는 논리주소이고, 논리 주소 4번지의 물리 주소는 8196임

80386 CPU의 구조를 통해 논리 주소, 물리 주소, MMU 엿보기



컴파일과 논리 주소

- 컴파일러는 프로그램을 논리 주소로 컴파일
  - 컴파일 시점에 프로그램이 물리 메모리 몇 번지에 적재될지 알 수 없음
  - 코드와 전역 변수들을 0번지에서부터 시작하는 논리 주소에 할당
- 응용프로그램 적재 시
  - 운영체제는 프로그램을 물리 메모리의 적절한 위치(비어있는)에 적재,
  - 논리 주소와 물리 주소의 매핑 테이블 생성
- 응용프로그램(프로세스) 실행 시
  - CPU가 인지하는 모든 주소는 논리 주소
    - 프로그램이 실행되면서 다루는 모든 주소는 논리 주소
    - CPU는 프로그램 내에 컴파일된 명령들을 다루며,
    - 명령들은 모두 논리 주소로 컴파일되어 있음
  - MMU는 CPU로부터 발생되는 논리 주소를 물리 주소로 변환
    - 매핑 테이블 참조
  - 동적 할당받은 메모리의 주소 역시 논리 주소
    - 물리 메모리가 할당되고 매핑 테이블에 논리 주소와 물리 주소의 항목 생성
  - 함수가 호출될 때 사용되는 스택 주소 역시 논리 주소

탐구 8-1 C 프로그램에서의 주소는 논리 주소인가 물리 주소인가?

C 프로그램 내에서 변수의 물리 주소를 알 수 있을까?
다음 C 프로그램은 전역 변수 n의 주소를 출력한다. 여기서 출력되는 변수 n의 주소 값은 논리 주소인가 물리 주소인가? 프로그램을 실행할 때마다 변수 n의 주소는 같을까 다를까?

logicaladdress.c

```
#include <stdio.h>
int n = 0;
int main() {
        printf("변수 n의 주소는 %p₩n", &n); // n의 주소 출력
}
```

```
$ gcc -o logical logicaladdress.c
$ ./logical
변수 n의 주소는 0x60103c
$ ./logical
변수 n의 주소는 0x60103c
$ ./logical
변수 n의 주소는 0x60103c
$
```

전역 변수 n의 주소는 논리 주소이다.
실행할 때마다 변수 n의 주소는 같다.
왜냐하면 논리 주소이기 때문이다.

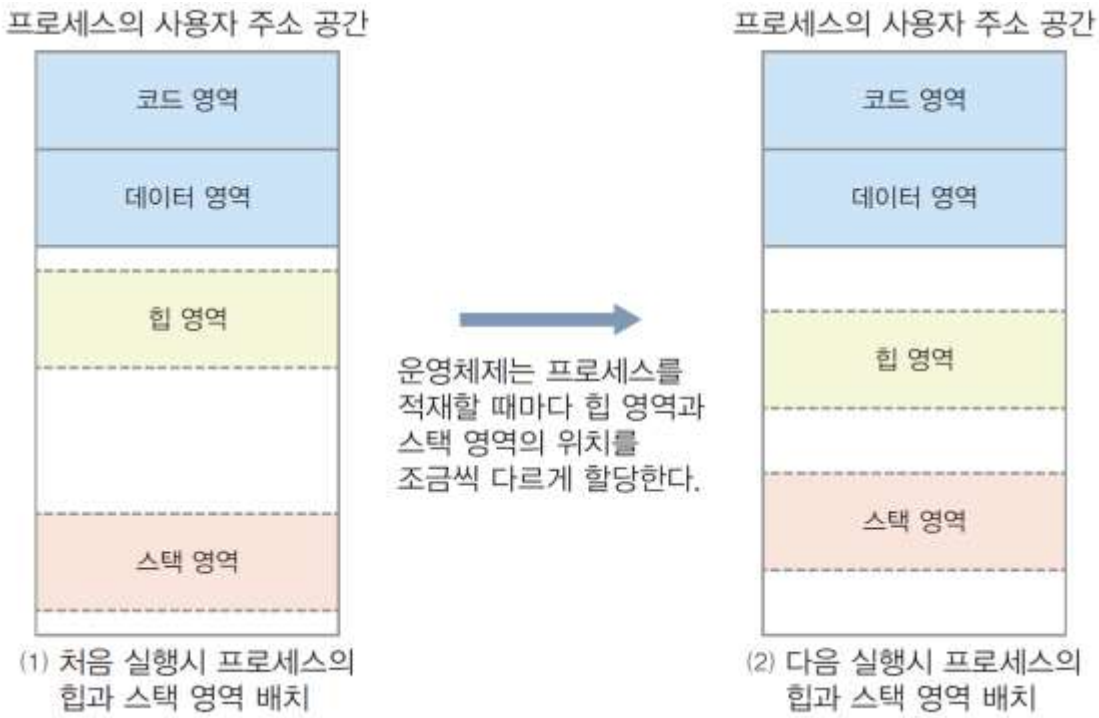
[주의]
탐구 8-1을 CoCalc 온라인 터미널에서 실행하면 실행 결과가 매번 다르게 출력된다. 그 이유는 CoCalc 온라인 터미널의 리눅스가 실행프로그램의 메모리 보호 기능 PIE를 적용하고 있기 때문이다.
다음과 같이 –no-pie 옵션으로 컴파일하면 실행파일의 PIE 기능을 해제하며, 그 후 실행시키면 변수 n의 주소는 동일하게 출력된다.

```
$ gcc –no-pie –o logical logicaladdress.c
$ ./logical
변수 n의 주소는 0x404030
$ ./logical
변수 n의 주소는 0x404030
$
```

* ASLR과 PIE에 관한 좀 더 자세한 내용은 생능출판사 홈 페이지의 제공 자료에 포함된 문서(**탐구 8-1의 실행에 있어 ASLR과 PIE 메모리 보호.pdf**)를 참고하라.

Tip. ASLR(Address Space Layout Randomization)

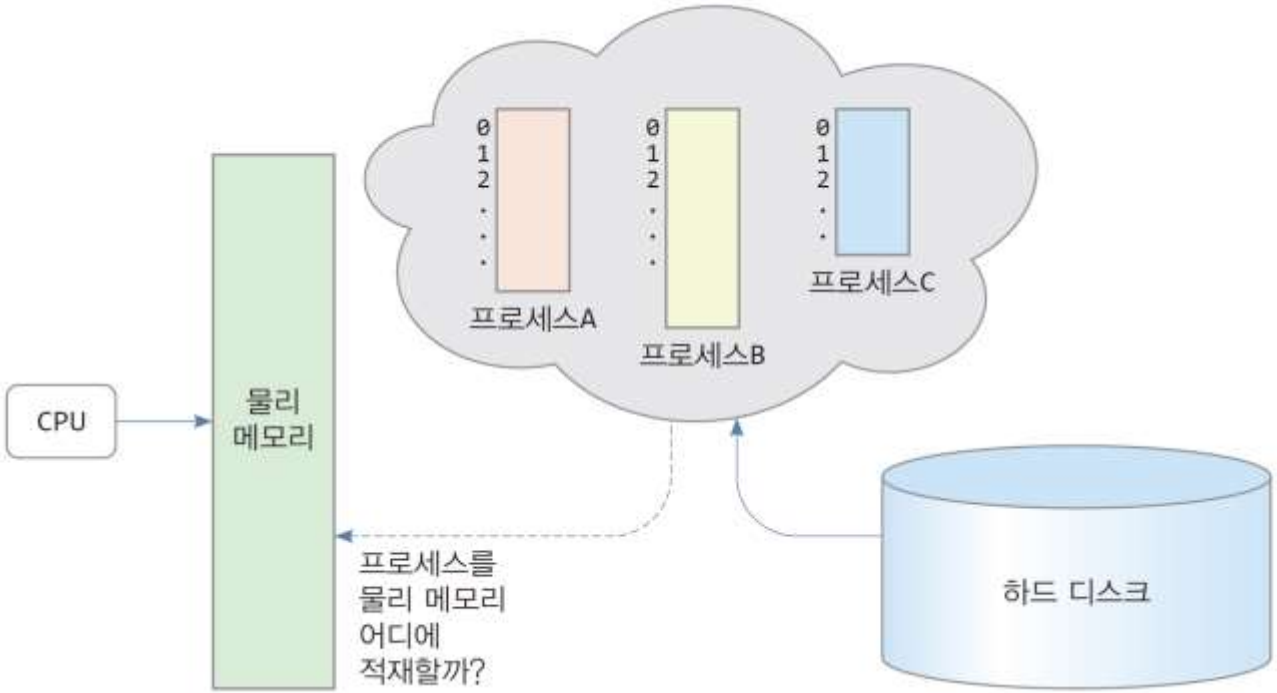
- ASLR
  - 해커들의 메모리 공격에 대한 대비책, 2001년경 도입, 오늘날 대부분의 운영체제가 활용
  - 주소 공간 랜덤 배치
    - 프로세스의 주소 공간 내에서 스택이나 힙, 라이브러리 영역의 랜덤 배치
    - 실행할 때마다 이들의 논리 주소가 바뀌게 하는 기법 -> 실행할 때마다 함수의 지역 변수와 동적 할당 받는 메모리의 논리 주소가 바뀜
    - 하지만, 코드나 전역 변수가 적재되는 데이터 영역의 논리 주소는 바뀌지 않음 PTE는 바뀜



3. 물리 메모리 관리

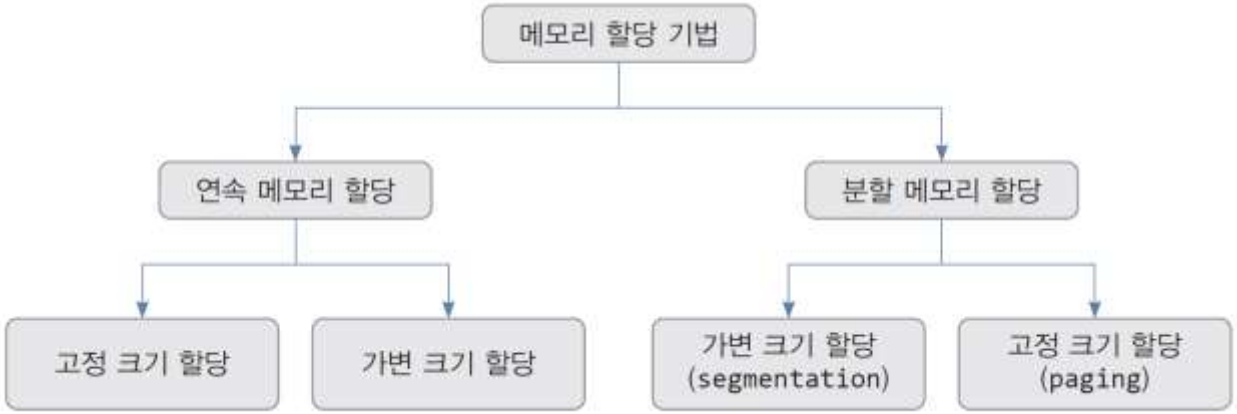
메모리 할당(memory allocation)

- 메모리 할당
  - 운영체제가 새 프로세스를 실행시키기거나 실행 중인 프로세스가 메모리를 필요로 할 때, 물리 메모리 할당
  - 프로세스의 실행은 할당된 물리 메모리에서 이루어짐
    - 프로세스의 코드(함수), 변수, 스택, 동적 할당 공간 액세스 등

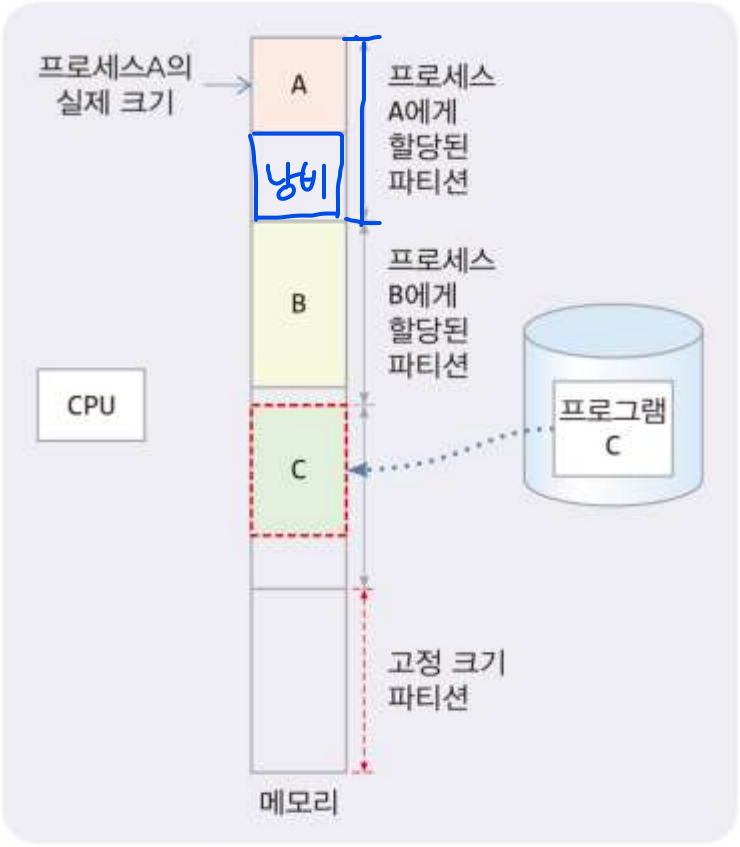


메모리 할당 기법

- 연속 메모리 할당
  - 프로세스별로 연속된 한 덩어리의 메모리 할당
  - 고정 크기 할당
    - 메모리를 고정 크기의 파티션으로 나누고 프로세스당 하나의 파티션 할당
    - 파티션의 크기는 모두 같거나 다를 수 있음
    - 메모리가 파티션들로 미리 나누어져 있기 때문에 고정 크기 할당이라고 부름
  - 가변 크기 할당
    - 메모리를 가변 크기의 파티션으로 나누고 프로세스당 하나의 파티션 할당
- 분할 메모리 할당
  - 프로세스에게 여러 덩어리의 메모리 할당
  - 고정 크기 할당
    - 고정 크기의 덩어리 메모리를 여러 개 분산 할당,. 대표 방법 : 페이징(paging) 기법
  - 가변 크기 할당
    - 가변 크기의 덩어리 메모리를 여러 개 분산 할당. 대표 방법 : 세그먼테이션(segmentation) 기법

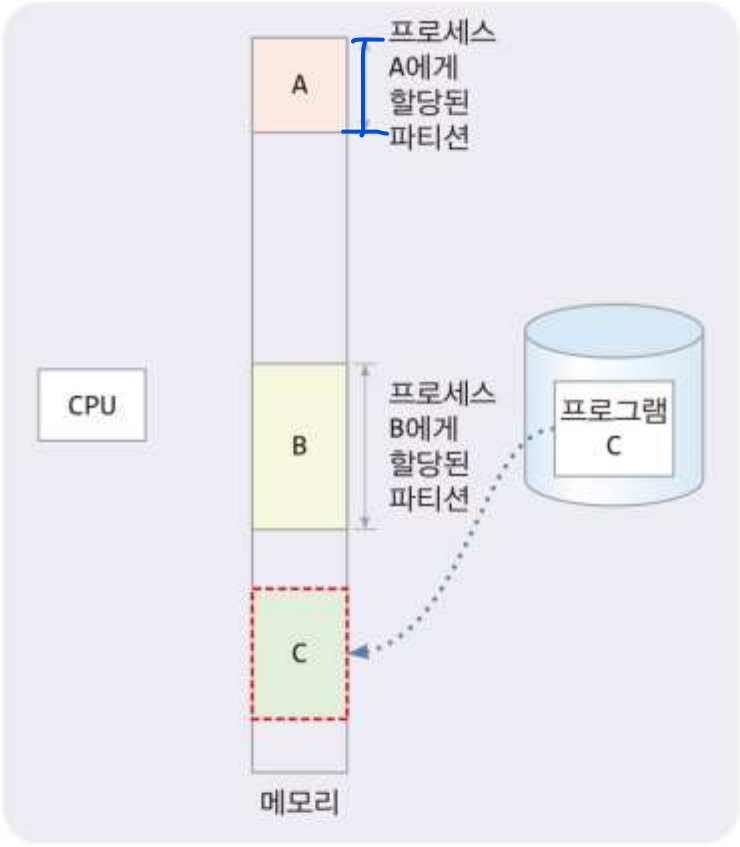


연속 메모리 할당



메모리를 고정 크기의 파티션으로 나누고
각 프로세스를 하나의 파티션에 배치

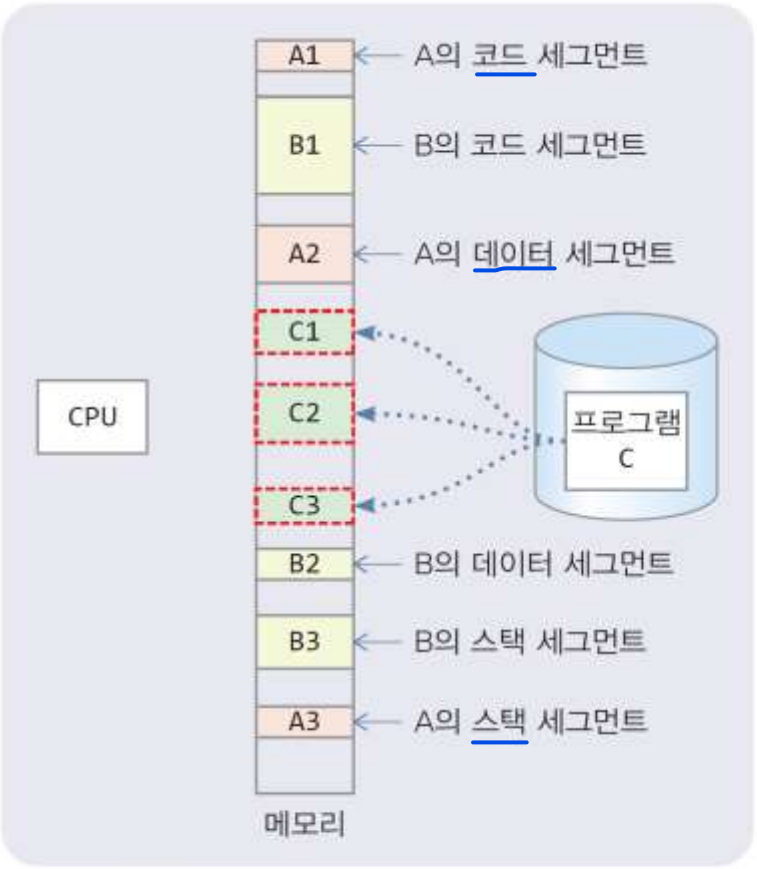
(a) 연속 메모리 할당 – 고정 크기



각 프로세스에게 자신의 크기만한
파티션 동적 할당

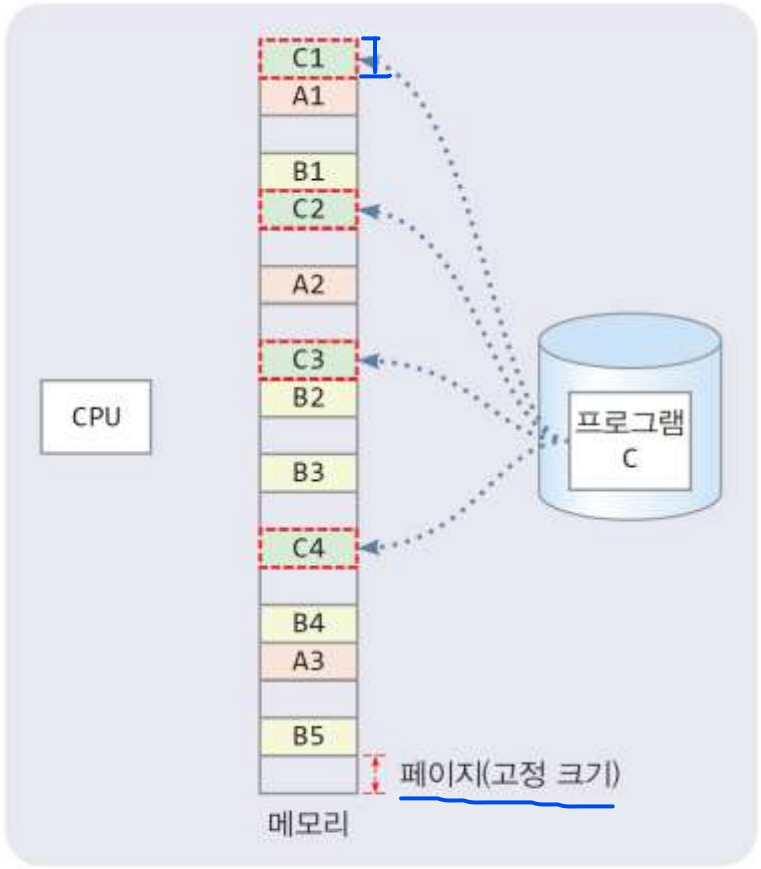
(b) 연속 메모리 할당 – 가변 크기

분할 메모리 할당



프로세스를 가변 크기의 세그먼트들로 분할 할당

(c) 분할 할당 – 세그먼테이션(segmentation)



프로세스를 고정 크기의 페이지들로 분할 할당

(d) 분할 할당 – 페이징(paging)

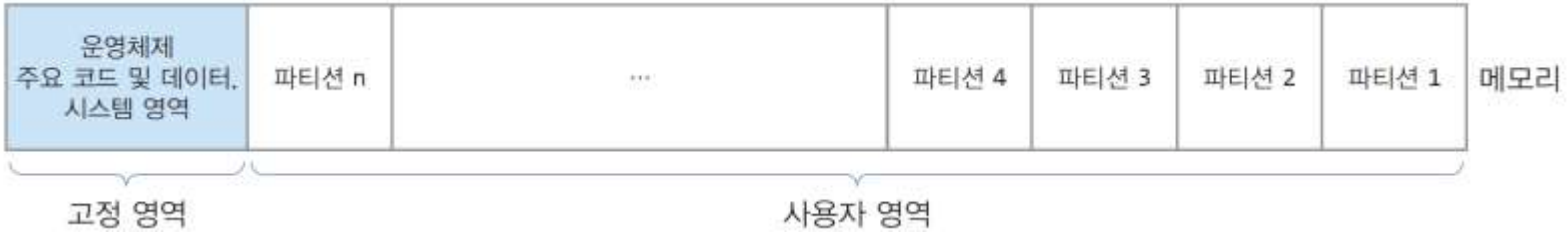
4. 연속 메모리 할당

연속 메모리 할당

- 프로세스를 1개의 연속된 공간에 배치
  - 메모리 전체를 여러 개의 파티션으로 분할,
  - 각 프로세스에게 한 개의 파티션 할당
- 연속 메모리 할당은 초기 운영체제에서 사용
  - MS-DOS와 같은 과거 운영체제
    - MS-DOS는 단일 사용자 단일 프로세스 시스템, 한 프로세스가 전체 메모리 독점
  - 고정 크기(fixed size partition) 할당 사례
    - IBM OS/360 MFT(Multiple Programming with a Fixed Number of Tasks)
    - 메모리 전체를 n개의 고정 크기로 분할. 프로세스마다 하나씩 할당
    - 수용가능 프로세스 수는 n개로 고정
    - 메모리가 부족할 때, 프로세스는 큐에서 대기
  - 가변 크기(variable size partition) 할당 사례
    - IBM OS/360 MVT(Multiple Programming with a Variable Number of Tasks)
    - 프로세스마다 프로세스 크기의 연속 메모리 할당
    - 수용가능 프로세스 수는 가변적임
    - 메모리가 부족할 때, 프로세스는 큐에서 대기
  - 가상 메모리 지원하지 않음

IBM 360의 연속 메모리 할당

- 고정 크기 할당
  - IBM OS/360 **MFT**(Multiple Programming with a Fixed Number of Tasks) 사례



- 가변 크기 할당
  - IBM OS/360 **MVT**(Multiple Programming with a Variable Number of Tasks) 사례

단편화

- 단편화(fragmentation)
  - 프로세스에게 할당할 수 없는 조각 메모리들이 생기는 현상, 조각 메모리를 홀(hole)이라고 부름
- 내부 단편화(internal fragmentation)
  - 할당된 메모리 내부에 사용할 수 없는 홀이 생기는 현상
    - 파티션보다 작은 프로세스(요구되는 메모리)를 할당하는 경우, 파티션 내에 홀 발생
    - IBM OS/360 MFT(Multiple Programming with a Fixed Number of Tasks) 사례

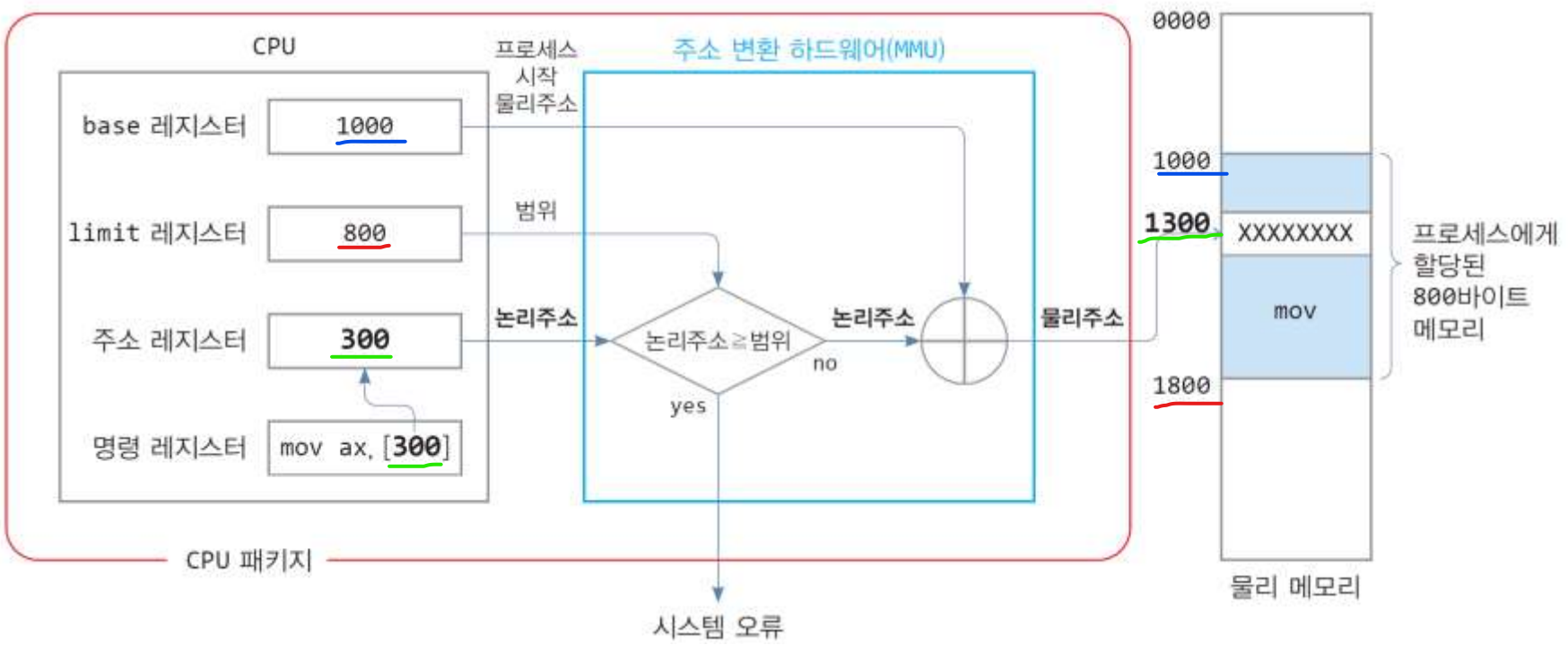
- 외부 단편화(external fragmentation)
  - 할당된 메모리들 사이에 사용할 수 없는 홀이 생기는 현상
    - 가변 크기의 파티션이 생기고 반환되는 여러 번의 과정에서 여러 개의 작은 홀 생성
    - 홀이 프로세스의 크기(요구되는 메모리 량)보다 작으면 할당할 수 없음
    - IBM OS/360 MVT(Multiple Programming with a Variable Number of Tasks) 사례

메모리 압축 기법 가능

연속 메모리 할당 구현

- 하드웨어 지원
  - CPU의 레지스터 필요
    - base 레지스터 : 현재 CPU가 실행중인 프로세스에게 할당된 물리 메모리의 시작 주소
    - limit 레지스터 : 현재 CPU가 실행중인 프로세스에게 할당된 메모리 크기
    - 주소 레지스터 : 현재 액세스하는 메모리의 논리 주소
  - 주소 변환 하드웨어(MMU) 필요 – 논리 주소를 물리 주소로 변환하는 장치
- 운영체제 지원
  - 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 할당에서 주소 변환과 메모리 보호



* 현재 실행 중인 프로세스는 물리 메모리 1000~1799 번지에 적재된 상황

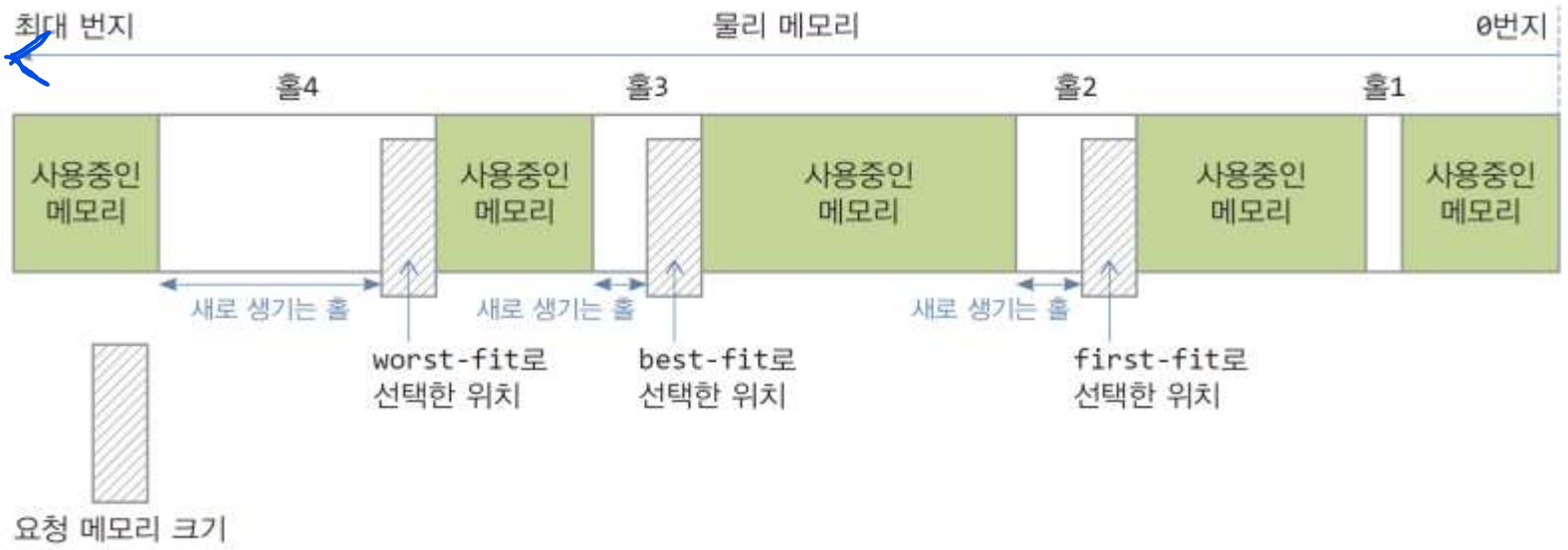
연속 메모리 할당의 장단점

- 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. 작을 홀들을 합쳐 충분한 크기의 메모리가 있음에도, 연속된 메모리를 할당할 수 없는 경우 발생
      - 메모리 압축 기법으로 해결

홀 선택 알고리즘/동적 메모리 할당

- 운영체제는 할당 리스트(allocation list) 유지
  - 할당된 파티션에 관한 정보 유지 관리
    - 할당된 위치, 크기, 비어 있는지 유무
- 할당 요청에 대해, 운영체제의 홀 선택 전략 3가지
  - first-fit(최초 적합)
    - 홀 리스트를 검색하여 처음으로 만나는, 요청 크기보다 큰 홀 선택
    - 할당 속도 빠름/단편화 발생 가능성
  - best-fit(최적 적합)
    - 홀 리스트를 검색하여 요청 크기를 수용하는 것 중, 가장 작은 홀 선택
    - 크기 별로 홀이 정렬되어 있지 않으면 전부 검색
  - worst-fit(최악 적합)
    - 홀 리스트를 검색하여 요청 크기를 수용하는 것 중, 가장 큰 홀 선택
    - 크기 별로 홀이 정렬되어 있지 않으면 전부 검색

3가지 홀 선택 알고리즘의 실행 사례



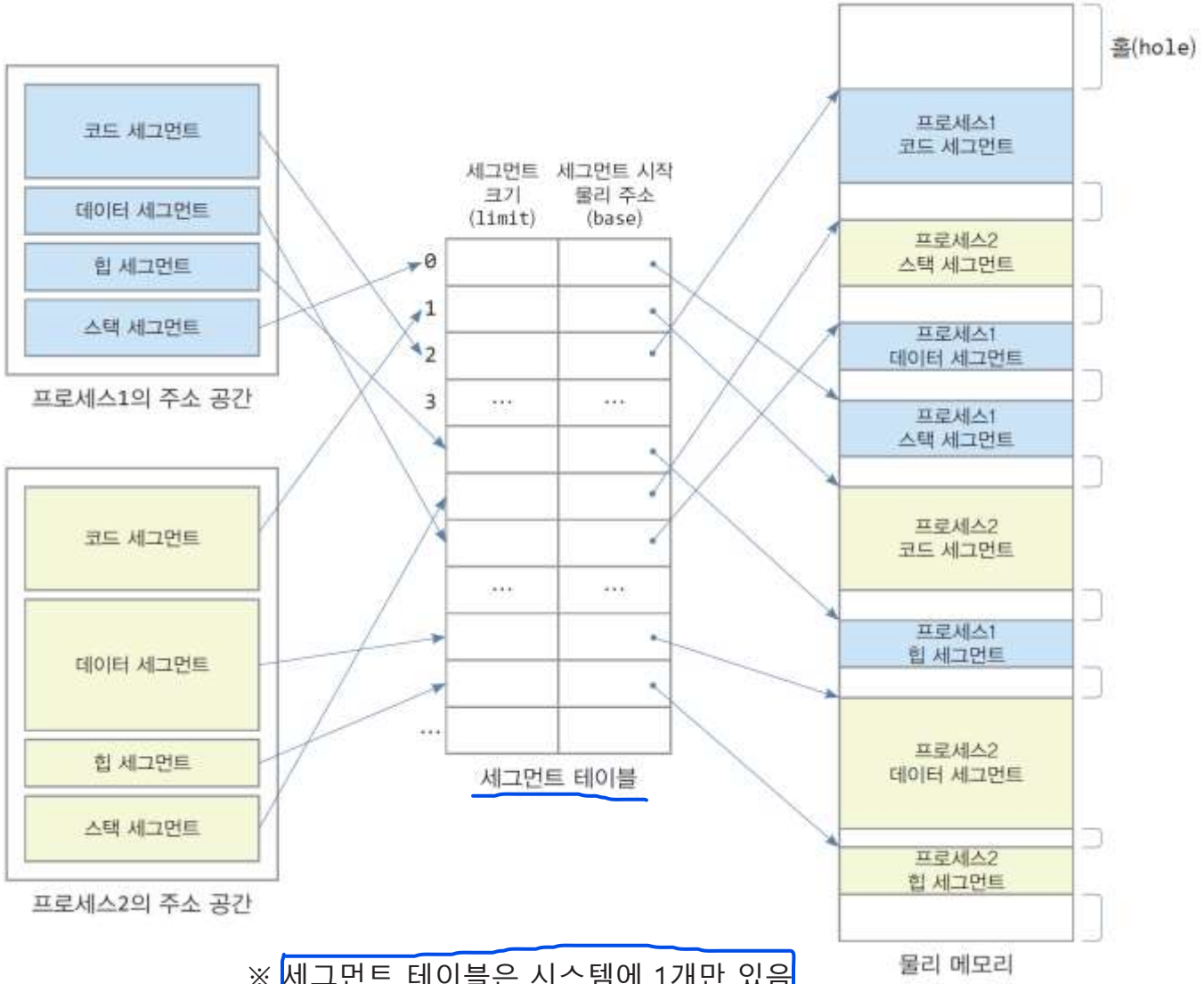
5. 세그먼테이션 메모리 관리

세그먼테이션(segmentation) 개요

페이징과 함께 쓰인다

- 세그먼트(segment)
  - 세그먼트는 프로그램을 구성하는 논리적 단위, 세그먼트마다 크기 다름
  - 일반적인 세그먼트 종류
    - 코드 세그먼트
    - 데이터 세그먼트
    - 스택 세그먼트
    - 힙 세그먼트
- 세그먼테이션 기법
  - 프로세스를 논리 세그먼트들로 나누고, 각 논리 세그먼트를 물리 메모리(물리 세그먼트)에 할당하는 메모리 관리 기법
  - 프로세스의 주소 공간
    - 프로세스의 주소 공간은 여러 개의 논리 세그먼트들로 구성
    - 각 논리 세그먼트는 물리 세그먼트에 매핑
    - 프로세스를 논리 세그먼트로 나누는 과정은 컴파일러와 링커에 의해 이루어짐
      - 컴파일러와 링커는 응용프로그램과 라이브러리의 코드를 모아 코드 세그먼트 구성, 전역변수들을 모아 데이터 세그먼트 구성
      - 운영체제 로더는 실행 파일에 구성된 각 논리 세그먼트를 물리 세그먼트에 할당, 논리 세그먼트 적재
  - 논리 세그먼트와 물리 세그먼트의 매핑
    - 시스템 전체에 1개의 세그먼트 테이블을 두고 논리 주소를 물리 주소로 변환
  - 외부 단편화 발생

논리 세그먼트와 물리 세그먼트 매핑

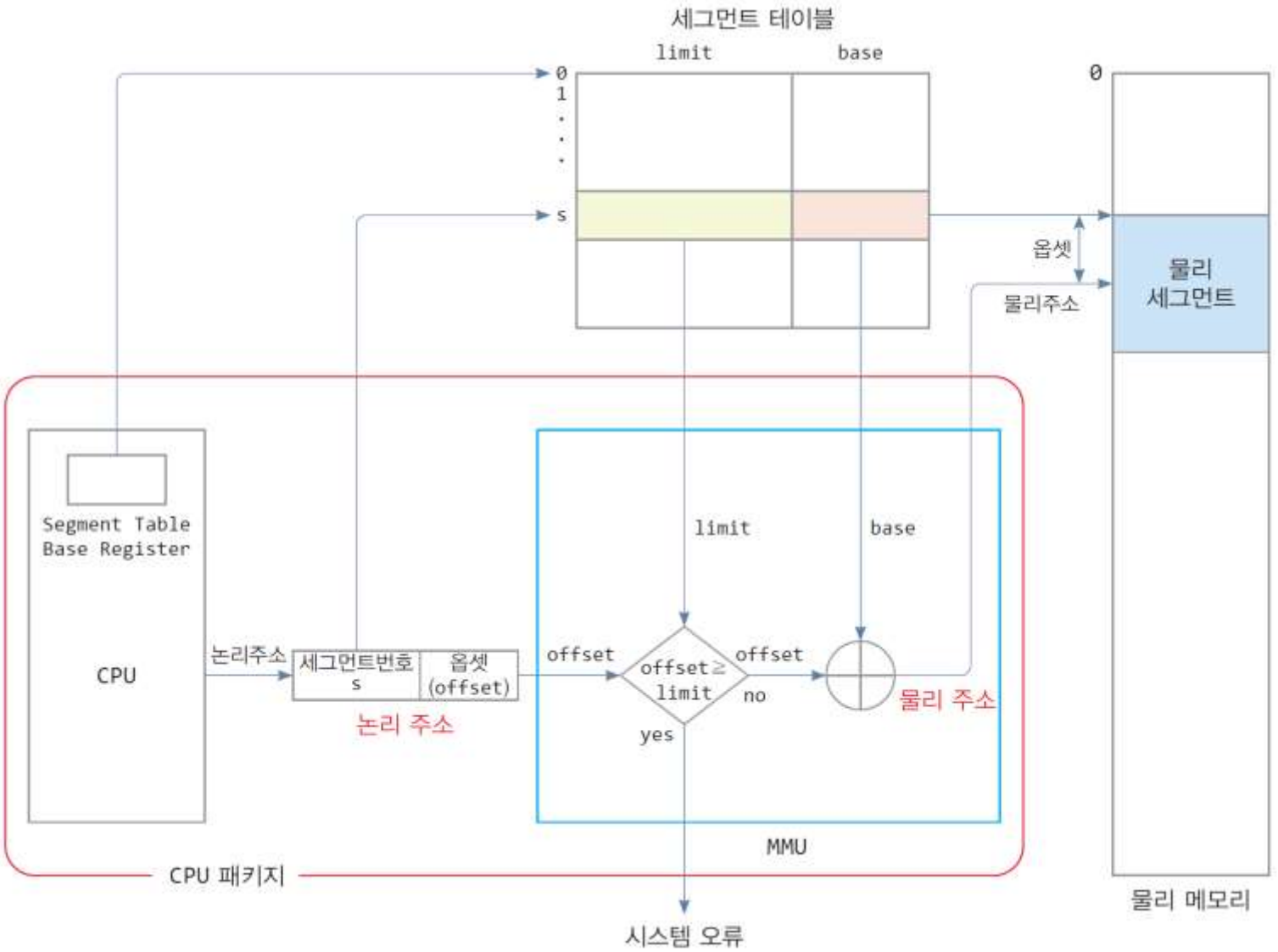


※ 세그먼트 테이블은 시스템에 1개만 있음

세그먼테이션의 구현

1. 하드웨어 지원
   - 논리 주소 구성 : [세그먼트 번호, 옵셋]
     - 옵셋 : 세그먼트내 상대 주소
   - CPU
     - 세그먼트 테이블의 시작 주소를 가리키는 레지스터(segment table base register) 필요
   - MMU 장치
     - 논리 주소를 물리 주소로 변환하는 장치
     - 논리 주소가 세그먼트 범위를 넘는지 판별(메모리 보호)
     - 논리 주소의 물리 주소 변환(메모리 할당)
   - 세그먼트 테이블
     - 메모리에 저장
     - 세그먼트별로 시작 물리 주소와 세그먼트 크기 정보
2. 운영체제 지원
   - 세그먼트의 동적 할당/반환 및 세그먼트 테이블 관리 기능 구현
     - 프로세스의 생성/소멸에 따라 동적으로 세그먼트 할당/반환
     - 물리 메모리에 할당된 세그먼트 테이블과 자유 공간에 대한 자료 유지
     - 컨텍스트 스위칭 때 CPU의 레지스터에 적절한 값 로딩
3. 컴파일러, 링커, 로더 지원
   - 사용자 프로그램을 세그먼트 기반으로 컴파일, 링킹, 로딩

세그먼테이션에서 논리 주소의 물리 주소 변환



단편화

99% 개선

- 외부 단편화 발생 이래서 페이징 기법 등장
  - 세그먼트들의 크기가 같지 않기 때문에 세그먼트와 세그먼트 사이에 발생하는 작은 크기의 홀
- 내부 단편화 발생 없음