Chapter

09

페이징 메모리 관리

1. 페이징 메모리 관리 개요
2. 페이징의 주소 체계
3. 페이지 테이블의 문제점과 TLB
4. 심화 학습: 페이지 테이블의 낭비 문제 해결

강의 목표

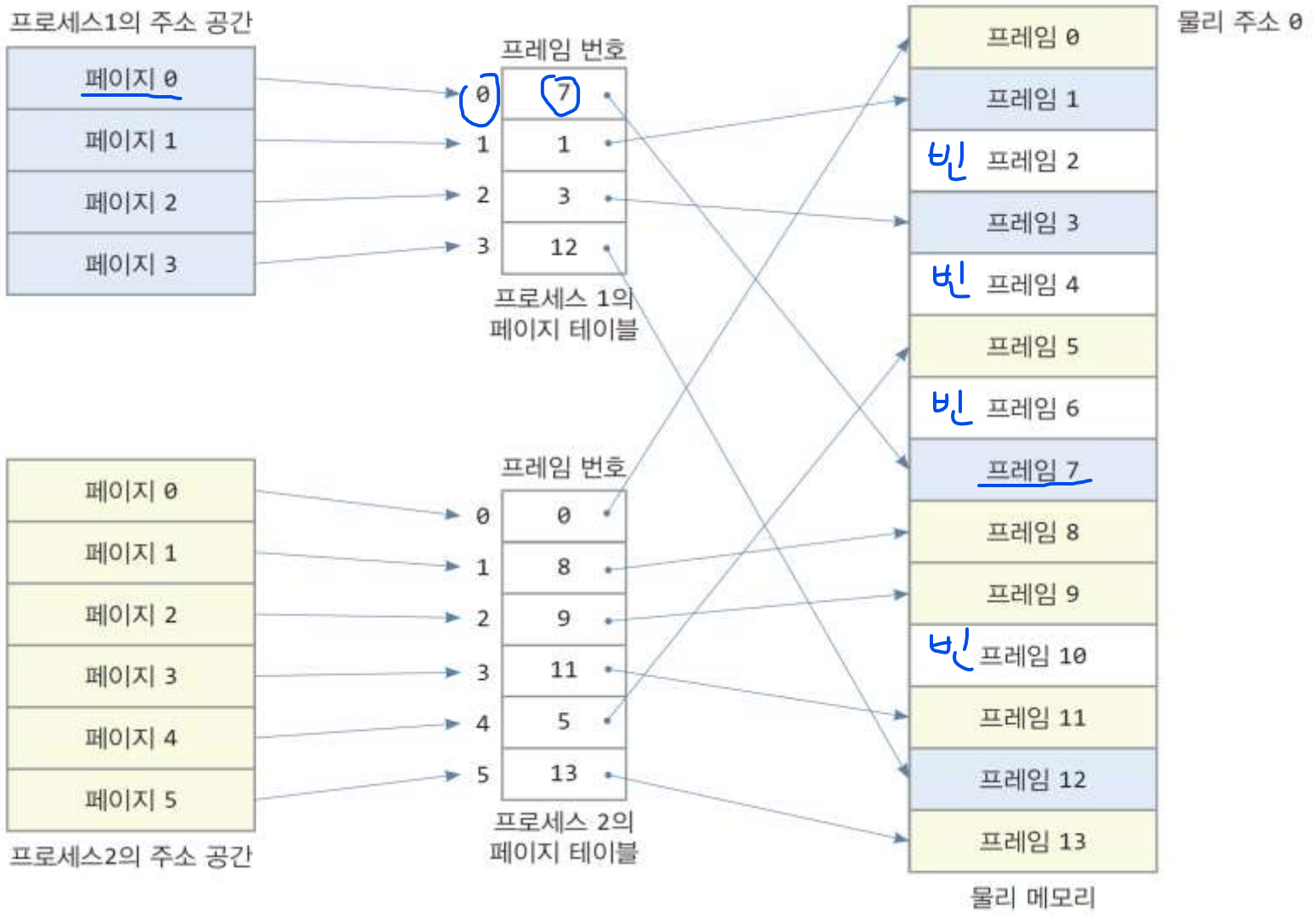
1. 페이지와 페이지 테이블로 이루어지는 페이징 메모리 관리의 개념을 이해한다.
2. 페이징에서 논리 주소와 물리 주소의 관계를 이해하고 주소 변환에 대해 안다.
3. 페이지 테이블로 인한 2가지 문제점을 이해하고 해결 방법을 안다.
4. TLB(Translation Look-aside Buffer)에 대해 알고 TLB가 프로그램의 메모리 액세스 성능을 향상시킴을 사례를 통해 이해한다.
5. 페이지 테이블의 낭비를 줄이는 방법으로 역페이지 테이블과 멀티 레벨 페이지 테이블에 대해 이해한다.

1. 페이징 메모리 관리 개요

페이징(paging) 개념

- 페이지와 프레임
  - 프로세스의 주소 공간을 0번지부터 동일한 크기의 페이지(page)로 나눔
    - 코드, 데이터, 스택 등 프로세스의 구성 요소에 상관없이 0번지부터 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 프로세스 마다 있다
- 페이징 기법
  - 프로세스의 주소 공간과 물리 메모리를 페이지 단위로 분할하고, 프로세스의 각 페이지를 물리 메모리의 프레임에 분산 할당하여 관리하는 기법
  - 프로세스의 주소 공간
    - 0에서 시작하여 연속적인 주소 공간
  - 프로세스마다 페이지 테이블 있음
  - 논리 주소의 물리 주소 변환 : MMU에 의해
  - 물리 메모리의 빈 프레임 리스트 관리 필요
    - 프레임 할당 알고리즘 : 빈 프레임 중에서 선택하는 알고리즘 필요
  - 내부 단편화 발생
  - 세그먼테이션보다 우수

논리 페이지와 물리 프레임 매핑



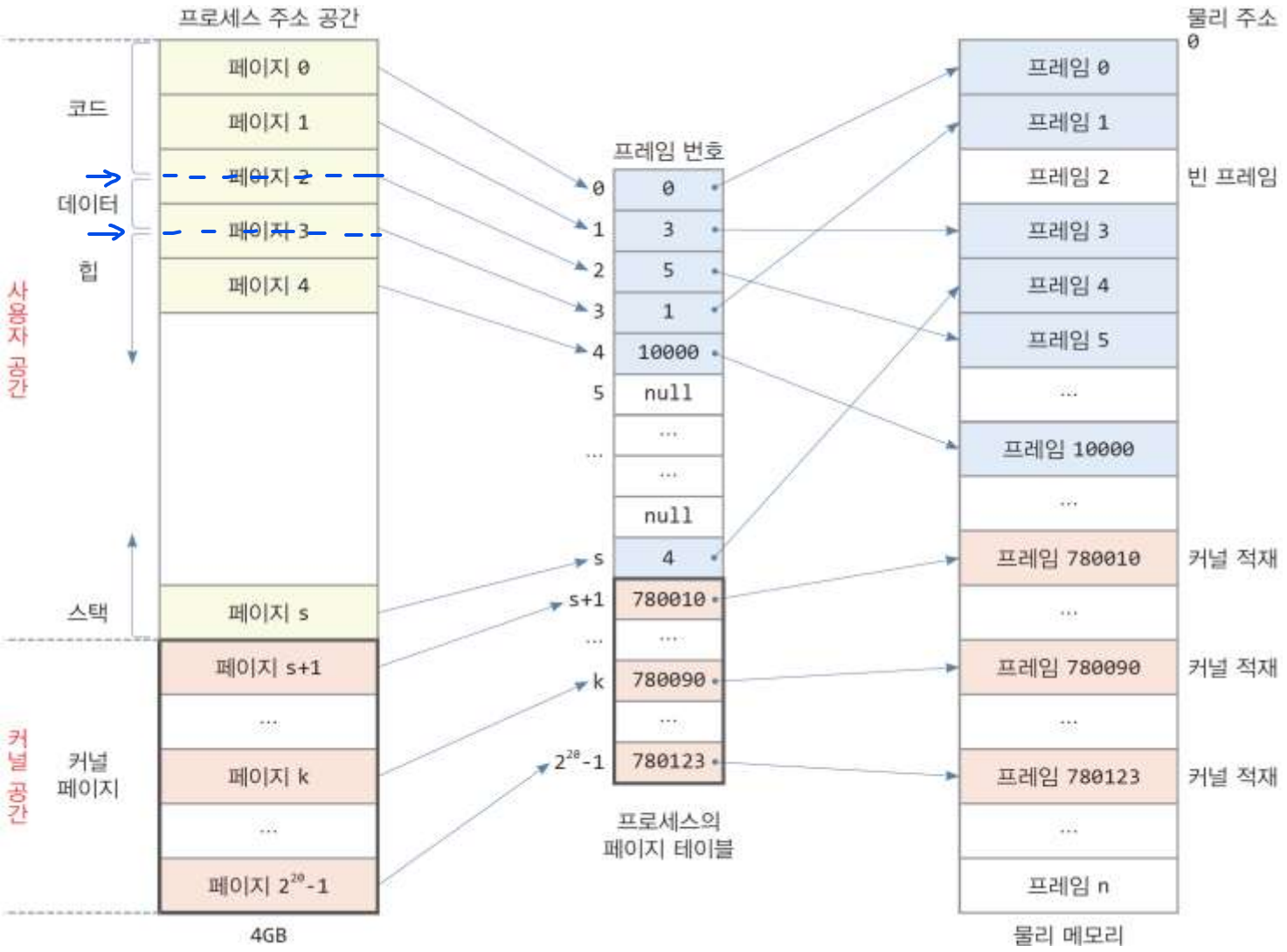
페이징의 우수성

- 용이한 구현
  - 메모리를 0번지부터 고정 크기의 페이지 단위로 단순 분할하기 때문
- 높은 이식성
  - 페이징 메모리 관리를 위해 CPU에 의존하는 것 없으므로,
  - 다양한 컴퓨터 시스템에 쉽게 이식 가능
- 높은 융통성
  - 시스템이나 응용에 따라 페이지 크기 다르게 설정 가능
- 메모리 활용과 시간 오버헤드면에서 우수
  - 외부 단편화 없음
    - 외부 단편화로 인한 메모리 낭비가 없고,
    - 홀 선택 알고리즘을 실행할 필요 없음
  - 내부 단편화는 발생하지만 매우 작음

페이지와 페이지 테이블(1)

- 다음 슬라이드의 사례로 설명
  - 4GB 주소 공간을 가지는 프로세스
  - 페이지 크기 4KB
  - 사례 프로세스
    - 코드 : 페이지 0 ~ 페이지 2에 걸쳐 있음
    - 데이터 : 페이지 2 ~ 페이지3에 걸쳐 있음
    - 힙 : 페이지 3 ~ 페이지 4에 걸쳐 있음
    - 스택 : 사용자 공간의 맨 마지막 페이지에 할당, 한 개 페이지 사용
  - 사례 프로세스는 6개 페이지 사용
    - 프로세스의 크기 : 6 x 4KB = 24KB
  - 페이지 테이블
    - 페이지 테이블은 주소 공간의 모든 페이지를 나타낼 수 있는 항목들을 포함
    - 현재 6개의 항목만 사용. 대부분의 항목은 비어 있음

32비트 CPU를 가진 컴퓨터에서 페이징 사례



페이지와 페이지 테이블(2)

- 프로세스가 동적 할당 받을 때(다음 슬라이드 설명)

```
char *p = (char*)malloc(200);        // 프로세스의 힙 영역에서 200 바이트 동적 할당
```

  - 200바이트 할당
    - 한 페이지(4KB) 할당
    - 논리 페이지 5 할당, 물리 프레임 2 할당
      - 페이지 5의 논리 주소 : 5*4KB = 20KB = 20*1024 = 20480 번지
      - 프레임 2의 물리 주소 : 2*4KB = 8192 번지
    - 페이지 테이블 항목 5에 물리 프레임 번호 2 기록
    - malloc(200)은 논리 주소 20480(페이지 번호 5)을 리턴

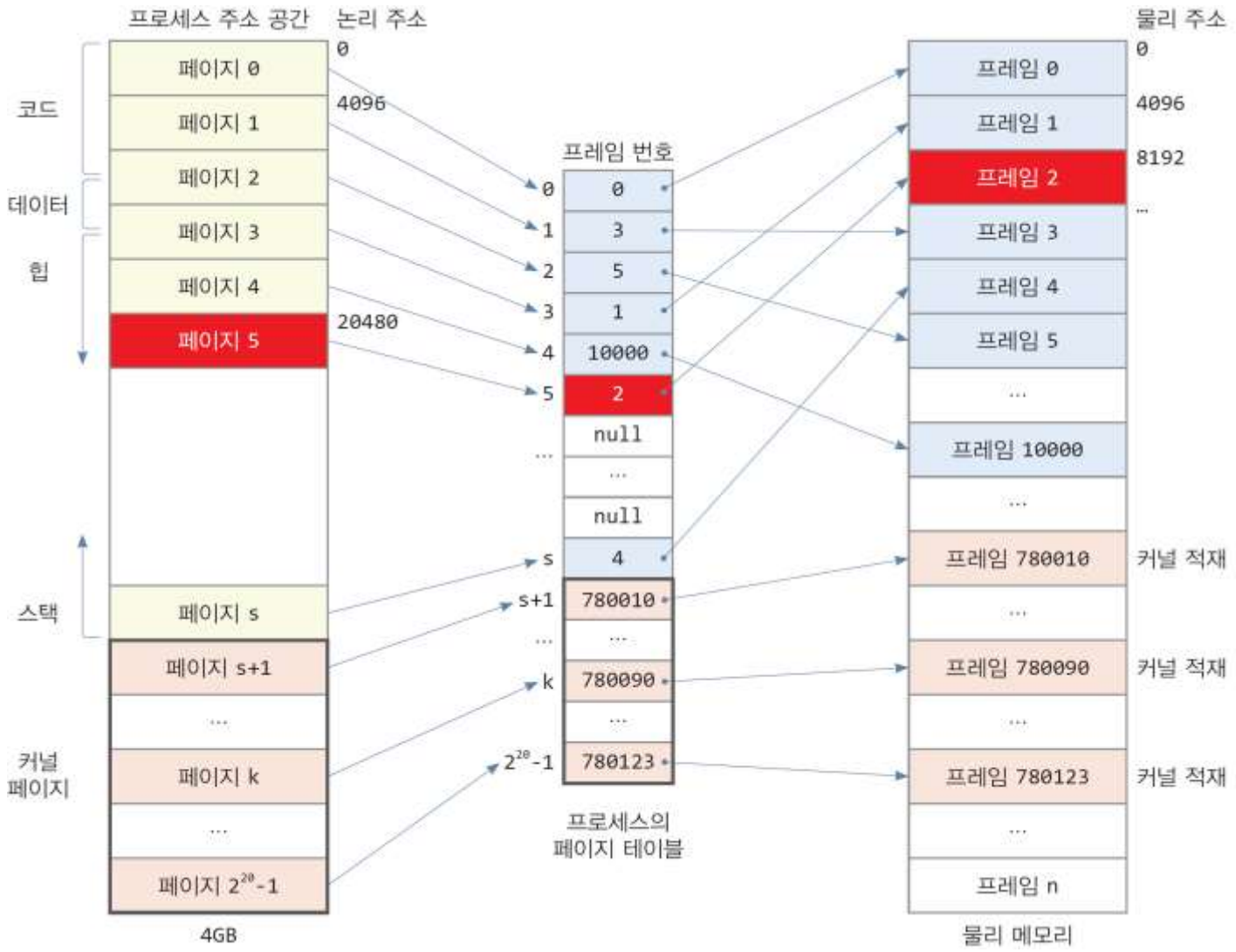
```
*p = 'a';
```

  - 프로세스 내에서 20480 번지에 'a'를 저장하는 코드
    - 논리 주소 20480이 MMU에 의해 물리 주소 8192로 바뀌어,
    - 물리 메모리 8192 번지에 'a' 저장

```
free(p);
```

  - 20480번지부터 200바이트 반환
    - 반환 후 페이지 5 전체가 비게 되므로, 페이지 5와 프레임 2가 모두 반환

프로세스가 200 바이트를 동적 할당 받을 때

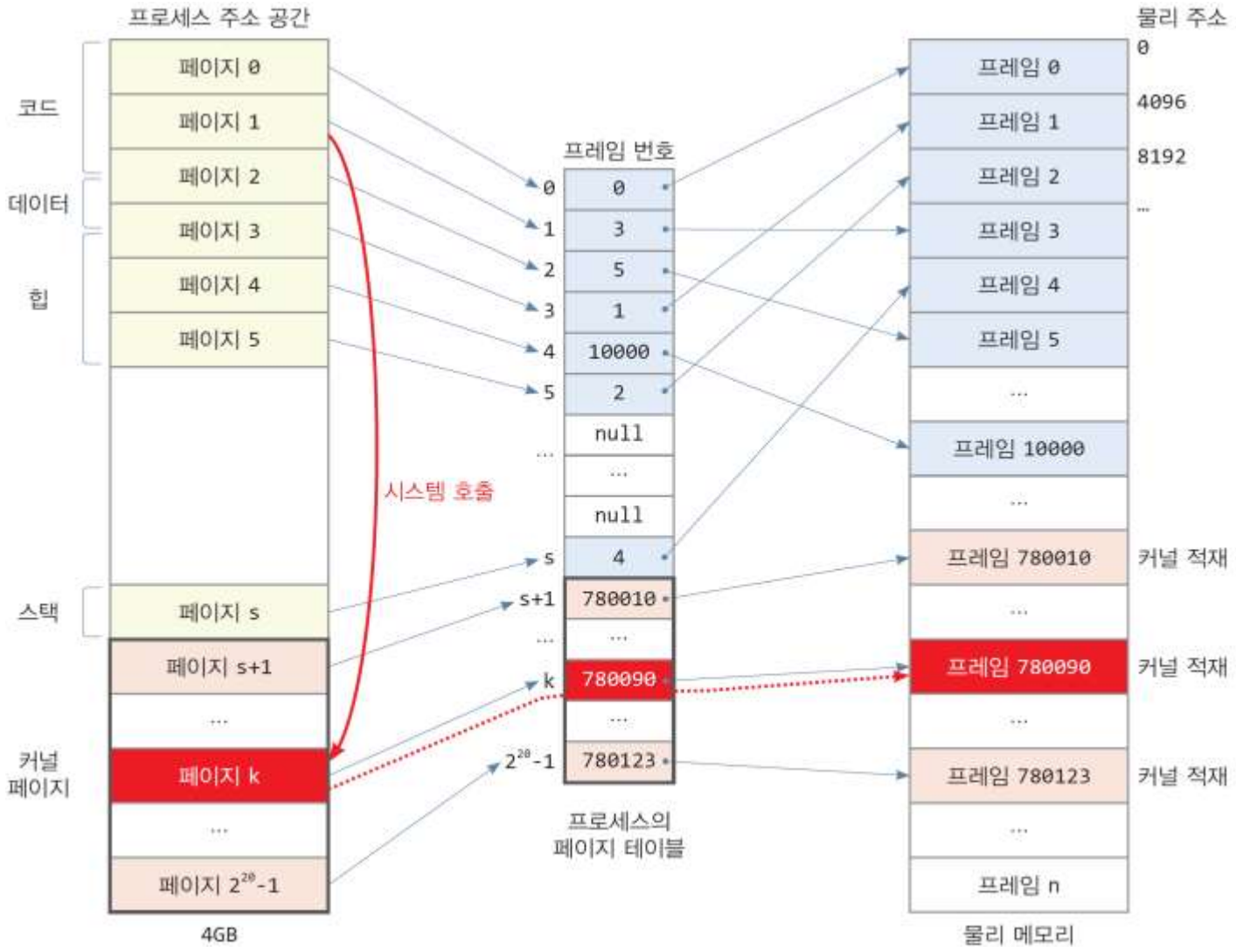


페이지와 페이지 테이블(3)

(논리적인 이야기)

- 프로세스가 시스템 호출을 실행할 때(다음 슬라이드)
  - 커널 공간의 페이지 k에 담긴 커널 코드 실행
  - 커널 코드 역시 논리 주소로 되어 있음
  - 현재 프로세스 테이블에서 페이지 k의 물리 프레임 780090을 알아내고 물리 프레임 780090에 적재된 커널 코드 실행
  - 중요 사항
    - 커널 코드도 논리 주소로 되어 있으며,
    - 시스템 호출을 통해 커널 코드가 실행될 때,
    - 현재 프로세스의 페이지 테이블을 이용하여 물리 주소로 변환

시스템 호출 시 프로세스의 페이지 테이블 활용



프로세스가 페이지 1의 코드에서 시스템 호출을 실행하는 경우 페이지 테이블을 거쳐 커널 코드 실행

페이지와 페이지 테이블에 대한 정리

- 32비트 CPU에서, 페이지 크기가 4KB인 경우
  - Q1. 물리 메모리의 최대 크기는 얼마인가?
    - 물리 주소의 범위는 $0 \sim 2^{32}-1$
    - 한 주소 당 한 바이트 크기이므로 물리 메모리의 최대 크기는 2^{32}=4GB
  - Q2. 프로세스의 주소 공간의 크기는 얼마인가?
    - 2^{32} 개의 주소들(한 주소당 1byte) 이므로, 총 4GB
    - 물리 메모리는 1GB, 2GB, 4GB 등 다양하게 설치될 수 있지만, 프로세스의 주소 공간은 물리 메모리 크기에 상관없이 4GB
  - Q3. 한 프로세스는 최대 몇 개의 페이지로 구성되는가
    - 4GB/4KB = $2^{32}/2^{12}$ = 2^{20}개 = 1M개 = 약 100만개
  - Q4. 프로세스당 하나의 페이지 테이블이 있다. 페이지 테이블의 크기는?
    - 페이지 테이블 항목 크기가 32비트(4B)라면(항목에는 프레임 번호 있음)
    - 4바이트*2^{20} = 2^{22}바이트 = 4MB 100만개 = 2^{20}
  - Q5. 그림 9-2의 상황에서 프로세스가 사용자 공간에서 사용하고 있는 크기는? 8P
    - 총 6개 페이지 사용
  - Q6. 응용프로그램이 하나의 프로세스라고 할 때, 응용프로그램의 최대 크기, 즉 개발자가 작성할 수 있는 프로그램의 최대 크기는?
    - 운영체제가 설정한 사용자 공간의 크기와 동일
  - Q7. 페이지 테이블 모양은?
    - 대부분의 항목이 비어 있는 희소 테이블(sparse table). 낭비가 심해 줄이는 기법 필요
  - Q8. 페이지 테이블은 어디에 존재하는가?
    - 메모리에 저장
  - Q9. 커널 코드는 논리 주소로 되어있는가, 물리 주소로 되어 있는가?
    - 커널 코드 역시 논리 주소로 되어 있음. 그러므로 커널 코드가 실행될 때 역시 물리 주소로 바뀌어야 하는데,
    - 이때 현재 프로세스의 페이지 테이블 사용

페이징에서의 단편화

- 외부 단편화 발생 없음
- 내부 단편화 발생
  - 스택이나 힙에 생성하는 페이지는 계속 변하므로 단편화 계산에서 제외한다면,
  - 프로세스의 마지막 페이지에만 단편화 발생
  - 단편화의 평균 크기 = 페이지의 ½ 크기

탐구 9-1 페이징 개념 확인

- 32비트 CPU에서, 페이지 크기 2KB, 설치된 물리 메모리 1GB, 프로세스 A는 사용자 공간에서 54321바이트를 차지한다고 할 때,
  - Q1: 물리 메모리의 프레임 크기는?
    - 2KB. 페이지 크기와 동일
  - Q2: 물리 메모리의 프레임 개수는?
    - 물리 메모리를 프레임 크기로 나누면 됨. 1GB/2KB = $2^{30}/2^{11}$ = 2^{19}개, 약 50만개
  - Q3: 프로세스의 주소 공간 크기와 페이지의 개수는?
    - 프로세스의 주소 공간 크기는 2^{32} = 4GB
    - 페이지의 개수 = $2^{32}/2^{11}$= 2^{21}개 =2M개 = 약 2백만개
  - Q4: 프로세스 A는 몇 개의 페이지로 구성되는가? 프로세스 A를 모두 적재하기 위한 물리 프레임의 개수는?
    - 프로세스 A의 실제 크기가 54321바이트이므로, 2KB(2048)로 나누면 54321/2048=26.5이므로 27개 페이지로 구성되며, 물리 프레임 역시 27개 필요
  - Q5: 페이지 테이블 항목 크기가 4바이트라고 할 때, 프로세스 A의 페이지 테이블 크기는?
    - 테이블 항목이 총 2^{21}개이므로 2^{21}*4바이트 = 2^{23}바이트 = 8MB
  - Q6: 페이징에서 단편화 메모리의 평균 크기는?
    - 프로세스의 코드와 데이터 연속되어 있으므로, 마지막 페이지에만 단편화가 생긴다. 단편화 메모리의 평균은 페이지의 반이므로 1KB
  - Q7: 페이지의 크기와 단편화의 관계는?
    - 페이지의 크기가 크면 단편화도 커진다.
  - Q8: 페이지의 크기와 페이지 테이블의 크기 관계는?
    - 페이지 크기가 크면 페이지 개수가 작아지고 페이지 테이블의 크기도 작아진다.

2. 페이징의 주소 체계

페이징의 논리 주소

- 논리 주소 구성
  - [페이지 번호(p), 옵셋(offset)]
    - 페이지 크기가 4KB(=2^{12})라면, 페이지 내 각 바이트 주소는 12비트
    - 옵셋 크기는 12비트
  - 32비트 논리 주소 체계에서, $2^{32} = 2^{20} \times 2^{12}$
    - 상위 20비트는 페이지 번호
    - 하위 12비트는 옵셋

20비트 12비트 20비트 12비트

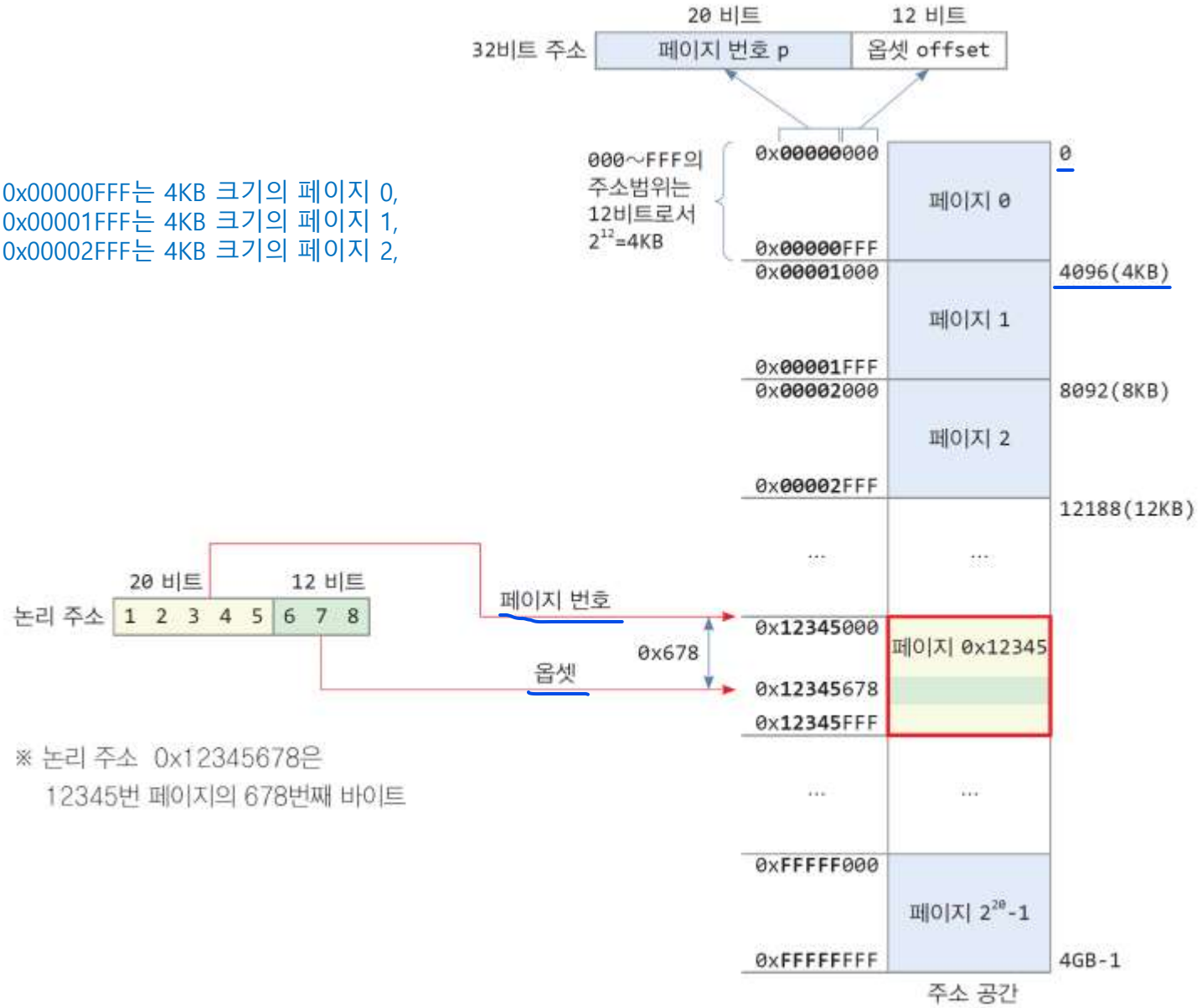
논리 주소 0x00000**000** ~ 0x00000**FFF** -> 페이지 0
논리 주소 0x00001**000** ~ 0x00001**FFF** -> 페이지 1
논리 주소 0x00002**000** ~ 0x00002**FFF** -> 페이지 2

.....

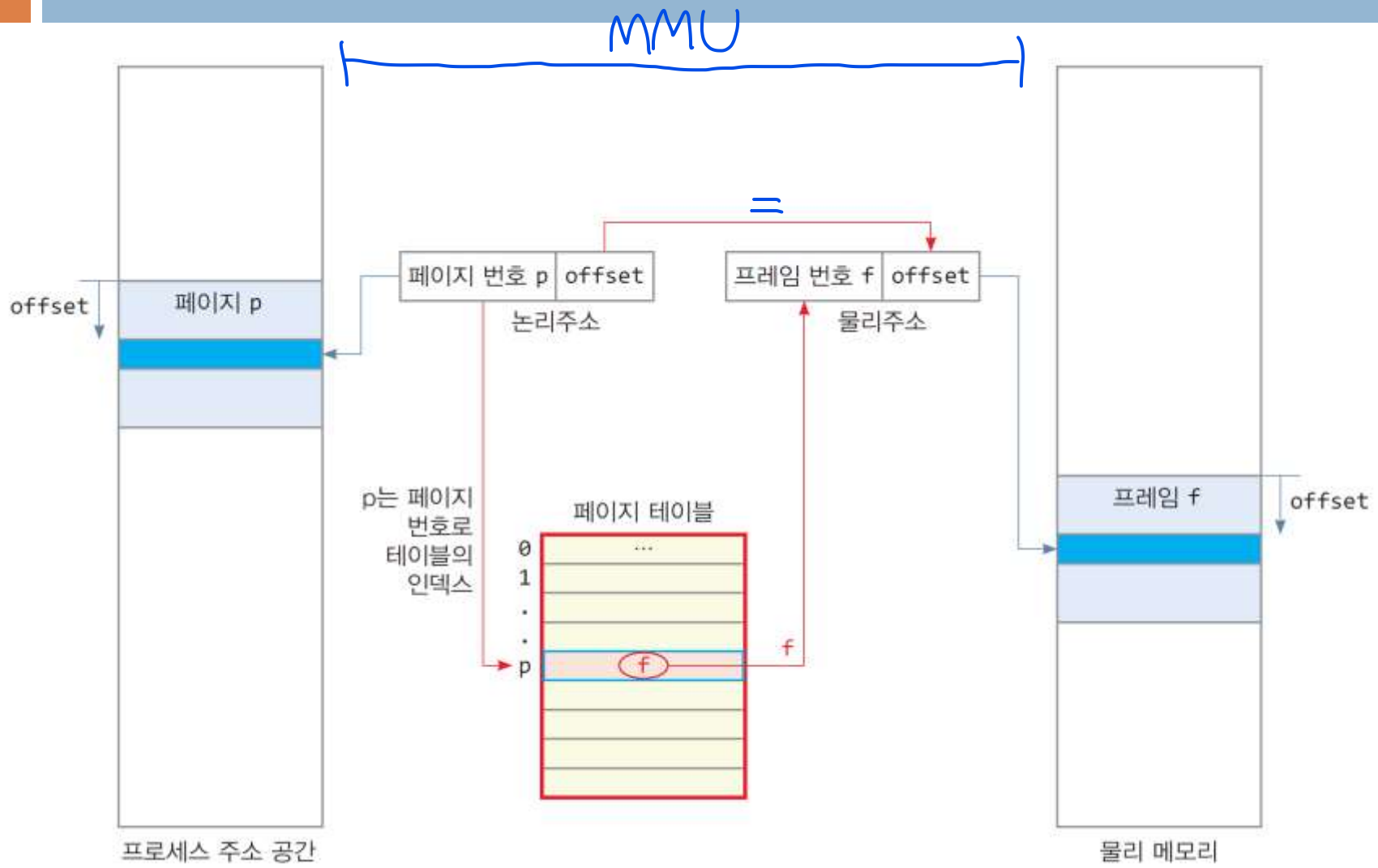
.....

32비트의 논리 주소와 페이지

0x00000000 ~ 0x00000FFF는 4KB 크기의 페이지 0,
0x00001000 ~ 0x00001FFF는 4KB 크기의 페이지 1,
0x00002000 ~ 0x00002FFF는 4KB 크기의 페이지 2,
.....



논리 주소의 물리 주소 변환 개념



페이징 구현

1. 하드웨어 지원
   - CPU의 지원
     - CPU에 페이지 테이블이 있는 물리 메모리 주소를 가진 레지스터 필요
       - Page Table Base Register(PTBR)
       - 이 레지스터는 운영체제에 의해 제어
   - MMU 장치
     - 논리 주소의 물리 주소 변환 장치
     - MMU는 CPU 패키지에 내장됨
2. 운영체제 지원
   - 물리 프레임의 동적 할당/반환
     - 물리 메모리의 빈 프레임 리스트 생성, 관리 유지
     - 프로세스의 생성/소멸에 따라 동적으로 프레임 할당/반환
   - 페이지 테이블 관리 기능 구현
     - 프로세스마다 페이지 테이블 생성, 관리 유지
     - 페이지 테이블이 저장된 물리 메모리 주소를 PCB에 저장
   - 프로세스 실행 시
     - 페이지 테이블의 물리 메모리 주소를 CPU의 PTBR(Page Table Base Register)에 적재

3.페이지 테이블의 문제점과 TLB

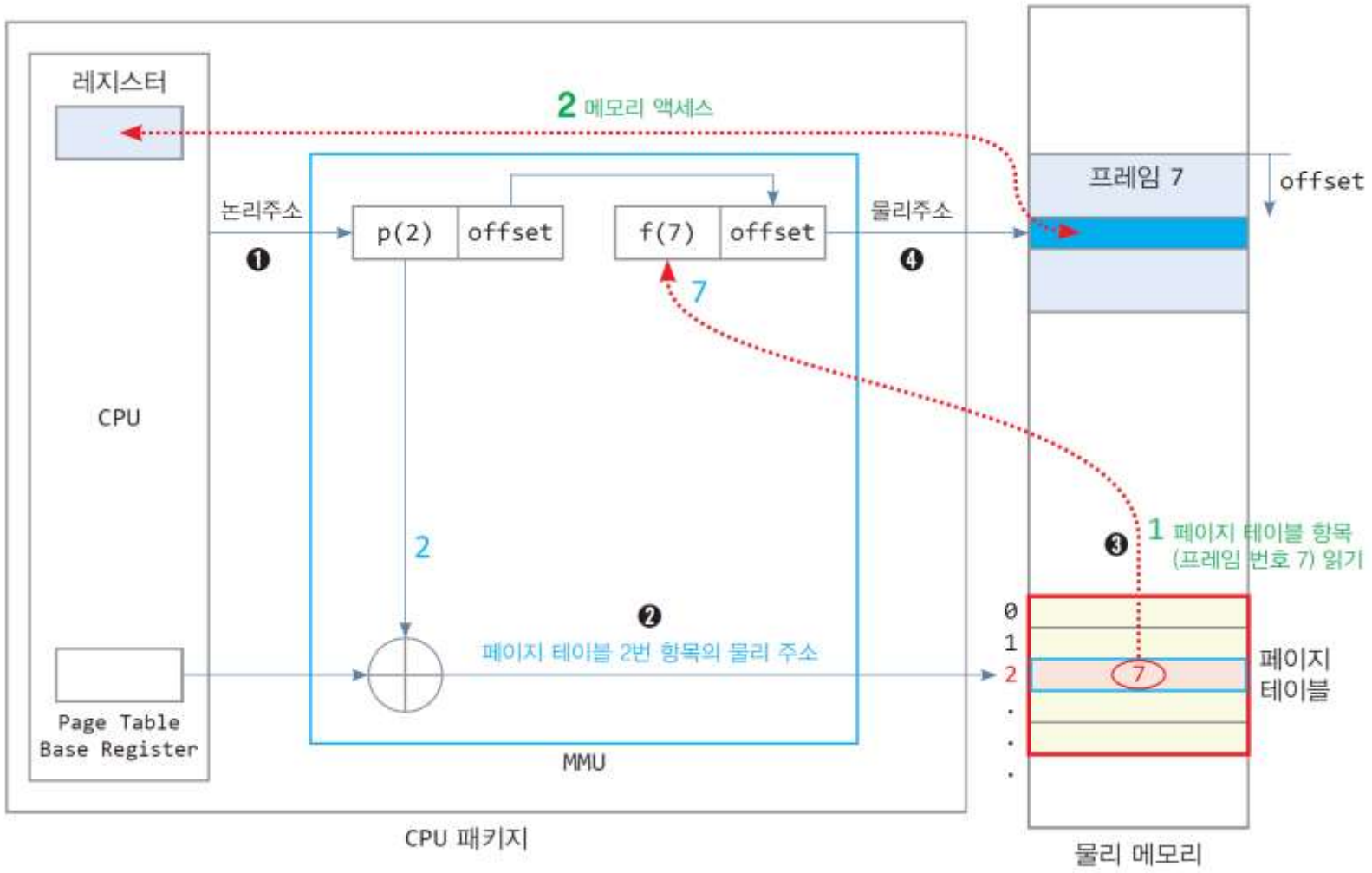
페이지 테이블의 문제점

- 문제점 2가지
  - **(문제1)** 1번에 메모리 액세스를 위한 **2번의 물리 메모리 액세스**
    - 페이지 테이블은 몇 MB의 비교적 큰 크기로, 메모리에 저장
    - CPU가 메모리를 액세스할 때 마다, 2번의 물리 메모리 액세스 -> 실행 속도 저하시킴 메모리가 일처리 느림, 그 동안 CPU는 놀게된다

      페이지 테이블 항목 읽기 1번 + 데이터 액세스 1번

    - TLB 사용으로 해결
  - **(문제2) 페이지 테이블의 낭비**
    - 프로세스의 실제 크기는 매우 작기 때문에,
    - 대부분의 페이지 테이블 항목이 비어 있는 문제
      - 페이지 테이블은 프로세스의 최대 크기를 기준으로 생성
    - '멀티레벨 페이지 테이블' 등의 방법으로 해결

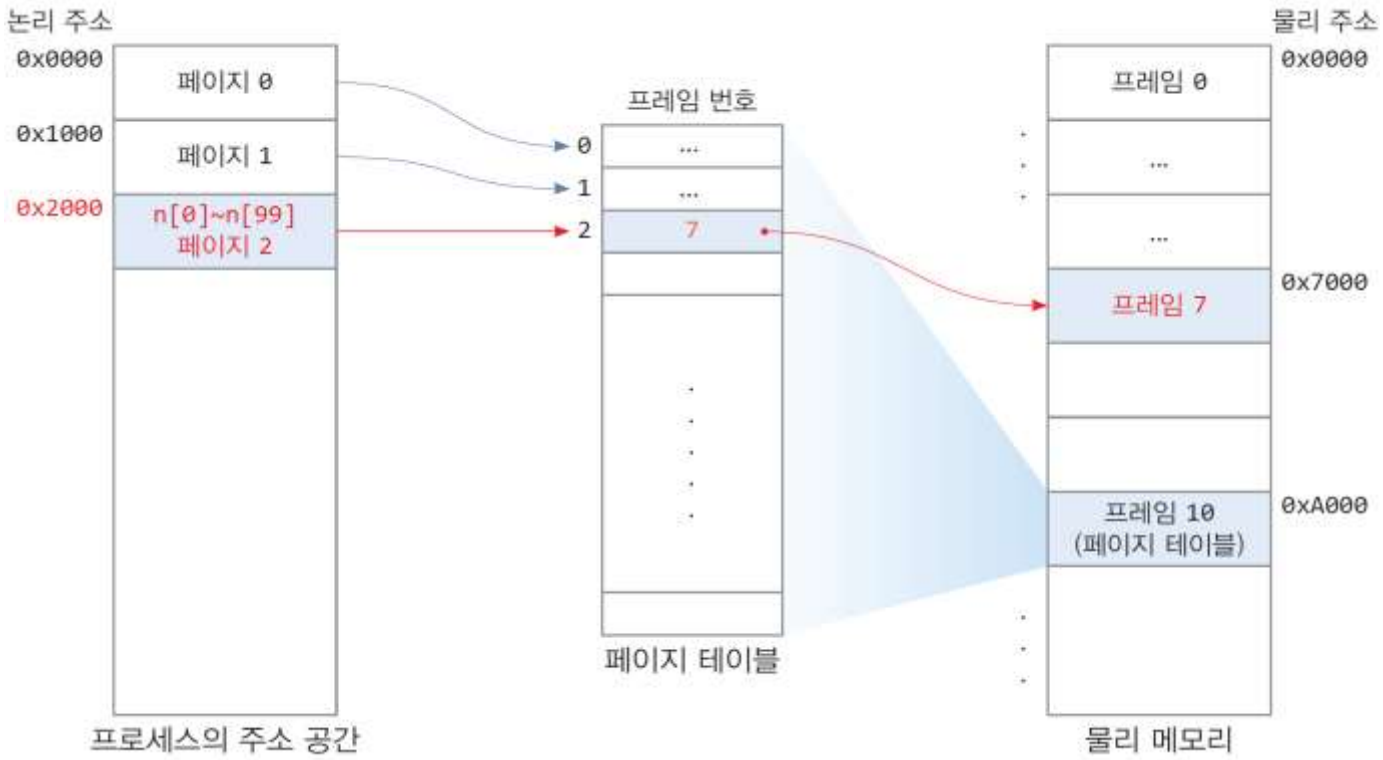
메모리 액세스 시 2번의 물리 메모리가 액세스되는 과정



탐구 9-2 C 프로그램이 실행될 때 메모리 액세스 과정 분석(1)

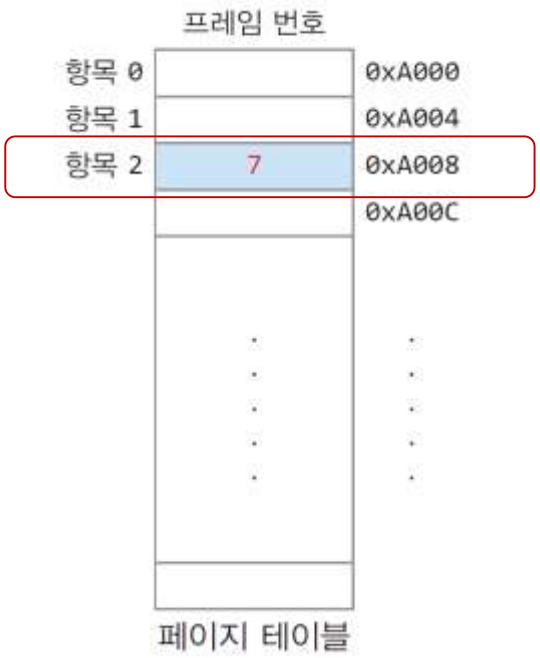

```
int n[100]; // 400바이트, 전역변수
int sum = 0; // 전역 변수
...
for(int i=0; i<100; i++)
   sum += n[i];
```

- 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 테이블은 물리 메모리 0xA000번지부터 시작



페이지 테이블의 2번 항목 주소

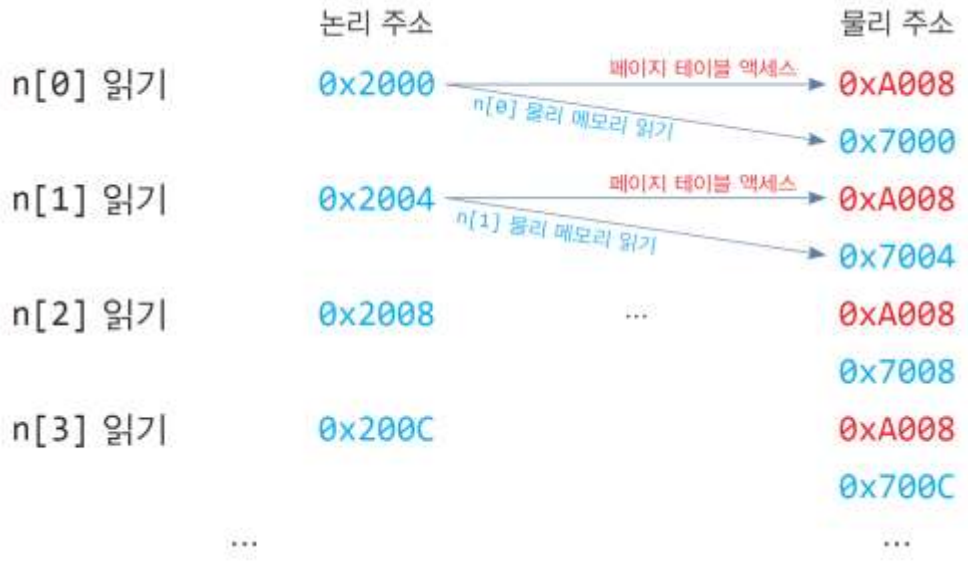
페이지 테이블의 2번 항목 주소 = 0xA008 번지



탐구 9-2 C 프로그램이 실행될 때 메모리 액세스 과정 분석(2)


```
int n[100]; // 400바이트, 전역변수
int sum = 0; // 전역 변수
...
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 테이블은 물리 메모리 0xA000번지부터 시작

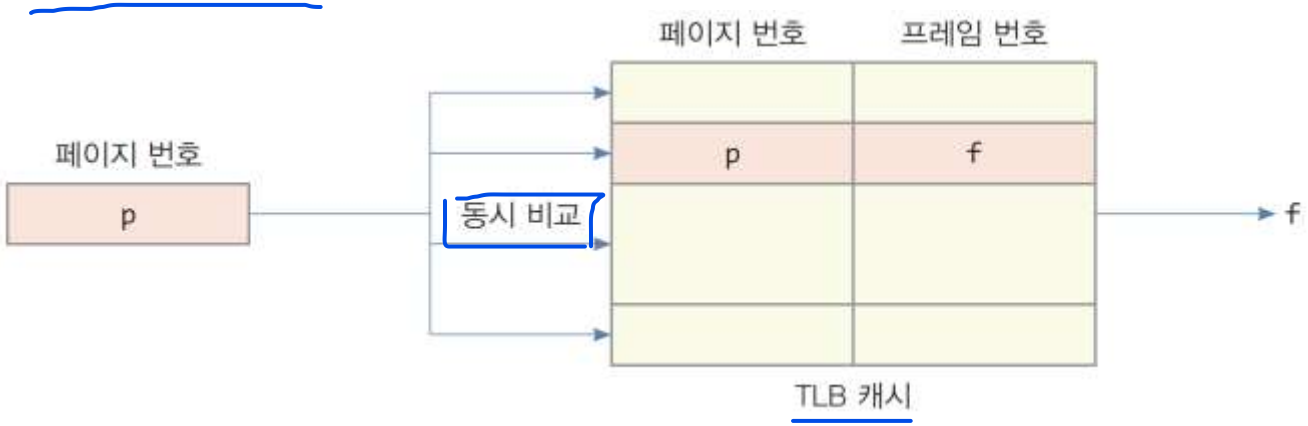


탐구 9-2를 통해 2가지 현상 발견

- n[i]의 값을 읽기 위해 2번의 메모리 액세스(페이지테이블 + n[i]의 물리 메모리)
- 배열 n[100]이 모두 한 페이지에 들어 있으므로, 배열 n[100]을 읽는 동안 하나의 페이지 테이블 항목(0xA008) 액세스

TLB를 이용한 2번의 물리 메모리 액세스 문제 해결

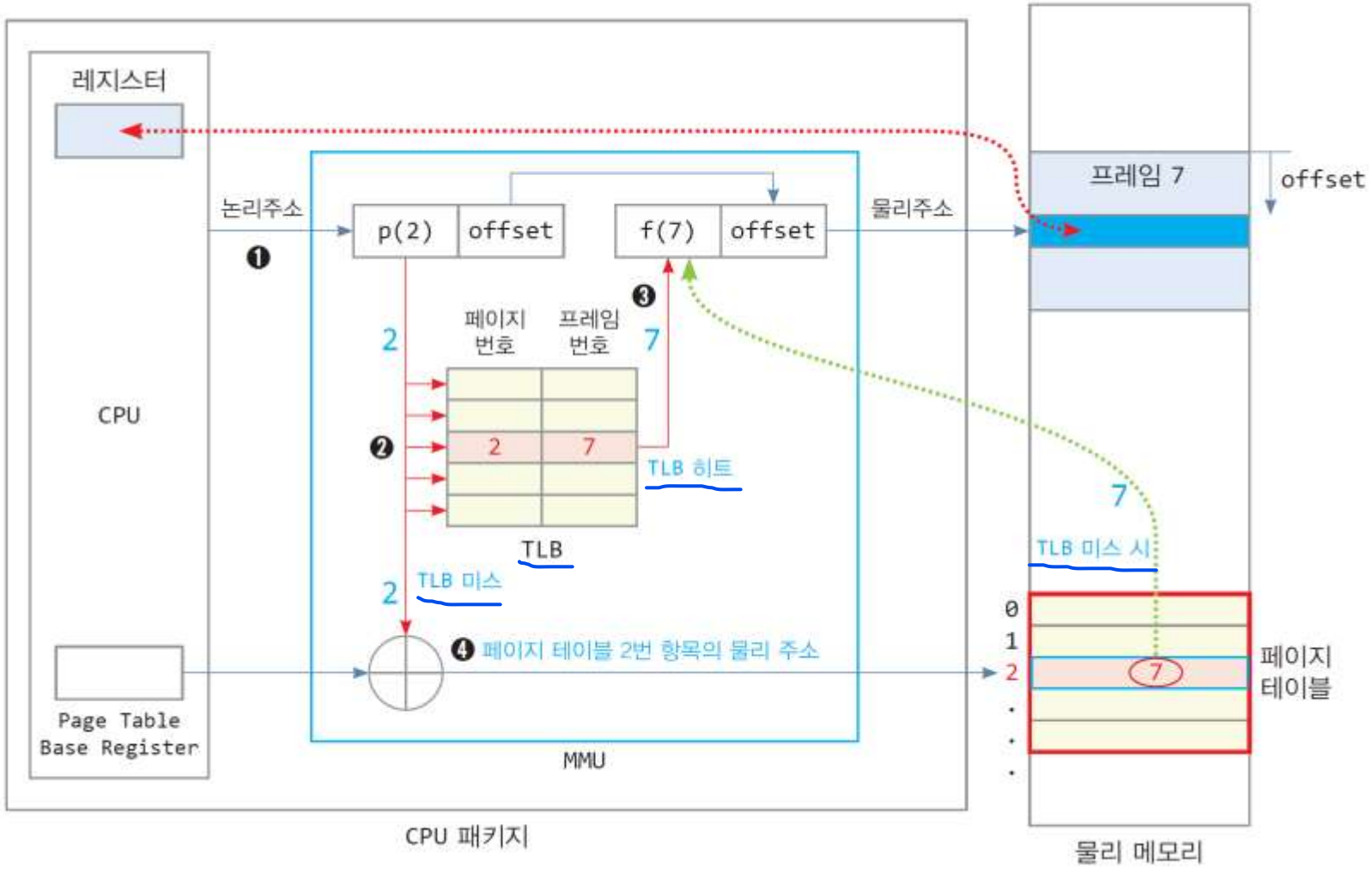
- 문제 해결 실마리
  - 논리 주소를 물리 주소로 바꾸는 과정에서 페이지 테이블을 읽어오는 시간을 없애거나 줄이는 기법
- TLB(Translation Look-aside Buffer) 사용
  - 주소 변환 캐시(address translation cache)로 불림
    - 최근에 접근한 '**페이지 번호와 프레임 번호**'의 쌍을 항목으로 저장하는 캐시 메모리
  - 위치
    - 현대 컴퓨터에서는 MMU 내에 존재
  - TLB 캐시의 구조와 특징
    - [페이지 번호 p, 프레임 번호 f]를 항목으로 저장
    - 페이지 번호로 전체 캐시를 동시에 고속 검색, 프레임 번호 출력
      - content-addressable memory, associative memory라고 불림
    - 고가, 크기 작음(64~1024개의 항목 정도 저장)



TLB를 활용한 메모리 액세스 과정

1. CPU로부터 논리 주소 발생
2. 논리 주소의 페이지 번호가 TLB로 전달
3. 페이지 번호와 TLB내 모든 항목 동시에 비교
   - TLB에 페이지 번호가 있는 경우, TLB hit
     - TLB에서 출력되는 프레임 번호와 offset 값으로 물리 주소 완성
   - TLB에 페이지 번호가 없는 경우, TLB miss
     - TLB는 miss 신호 발생
     - MMU는 페이지 테이블로부터 프레임 번호를 읽어와서 물리 주소 완성
     - 미스한 페이지의 [페이지번호, 프레임번호] 항목을 TLB에 삽입

TLB를 가진 경우 메모리 액세스 과정



탐구 9-3 TLB가 있는 경우 C 프로그램 실행 과정 분석


```
int n[100]; // 400 바이트, 전역변수
int sum = 0; // 전역 변수
...
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 주소가 0x7000(프레임 7)부터 시작
- 배열 n[100]의 크기는 400바이트이며 페이지 2에 모두 들어 있음
- 페이지 테이블이 메모리의 0xA000번지에서 시작

탐구 9-3을 통해 2가지 현상 발견

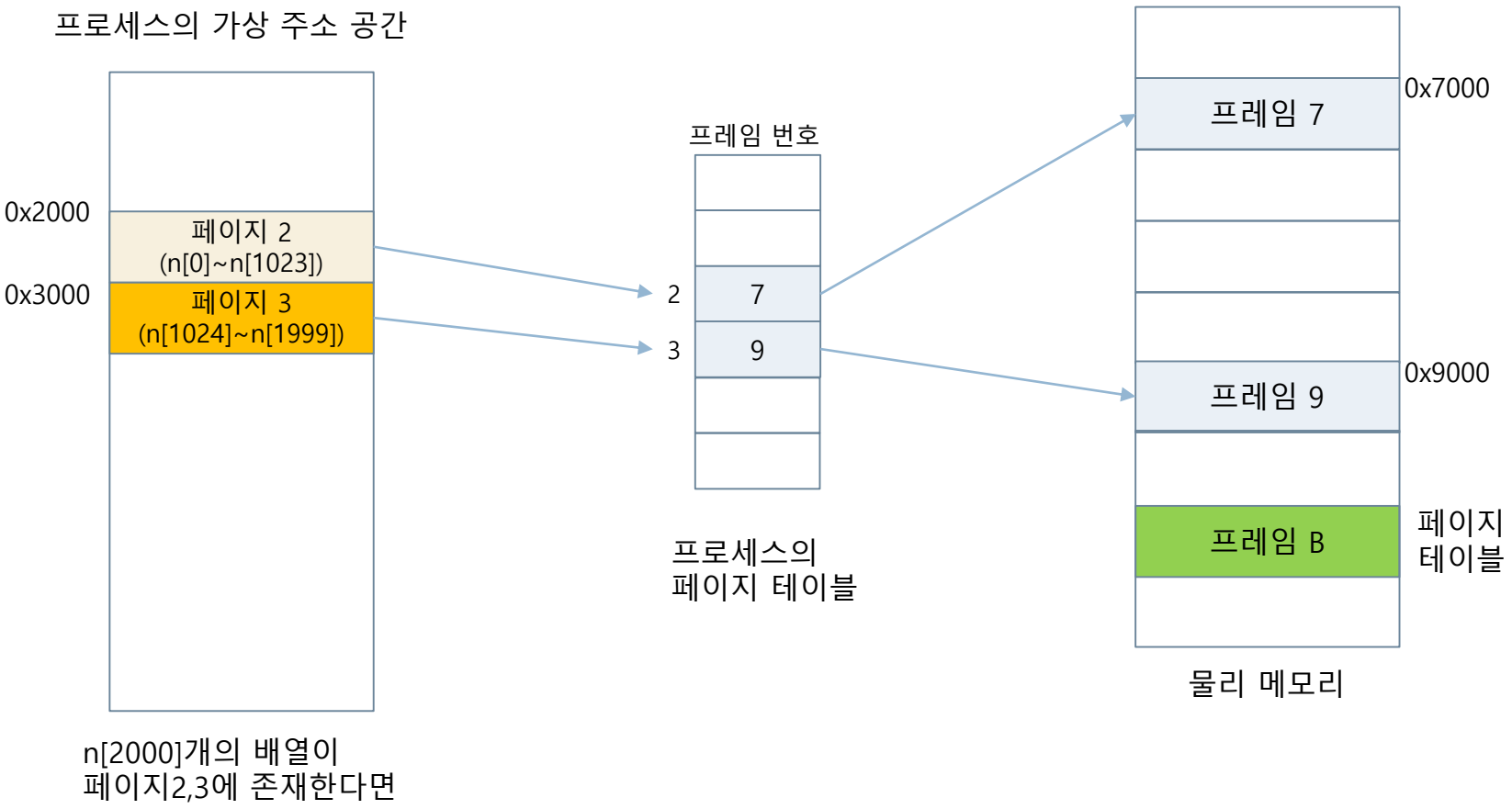
- TLB 미스가 발생하는 첫번째 경우에만 물리 메모리 2번 액세스(페이지테이블+n[i]의 물리 메모리)
- 동일한 페이지를 연속하여 액세스하는 동안 TLB hit 계속 발생 – 페이지테이블 액세스 횟수 줄여짐 – 성능 향상

탐구 9-4 배열이 2개의 페이지에 걸쳐 있는 경우 TLB 활용 사례

```
int n[2000]; // 8000 바이트, 전역변수
int sum = 0; // 전역 변수
...
for(int i=0; i<2000; i++)
    sum += n[i];
```

- 배열 n[2000]의 논리 주소는 0x2000부터 시작하여 페이지 2, 3에 걸쳐 존재
- 배열 n[2000]의 물리 주소는 0x7000부터 시작하여 프레임 7과 9에 나누어 할당
- 페이지 테이블이 메모리 0xA000번지에서 시작

탐구 9-4를 위한 참고 그림



TLB로부터 얻는 교훈

- TLB와 참조의 지역성
  - TLB는 참조의 지역성으로 인해 효과적인 전략임
  - TLB를 사용하면, 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 잦은 항목 교체(TLB 항목 수가 제한되기 때문)
- TLB 성능
  - TLB 히트율 높이기 -> TLB 항목 늘이기(비용과 trade-off)
  - 페이지 크기
    - 페이지가 클수록 TLB 히트 증가 -> 실행 성능 향상
    - 페이지가 클수록 내부 단편화 증가 -> 메모리 낭비
    - 페이지가 크면 장단점이 동시에 존재하므로 선택의 문제
    - 현대에서 페이지가 커지는 추세 : 디스크 입출력의 성능 향상을 위해
- TLB reach
  - TLB 도달 범위
    - TLB가 채워졌을 때, 미스없이 작동하는 메모리 액세스 범위
    - TLB 항목 수 * 페이지 크기

TLB를 고려한 컨텍스트 스위칭 과정 재정리

- 동일한 프로세스 내에서 다른 스레드로 스위칭되는 경우
  - TLB 항목들이 교체될 필요 없음
    - 동일한 프로세스의 주소 공간에서 실행되므로 동일한 프로세스 테이블 사용되기 때문

- 다른 프로세스의 스레드로 스위칭되는 경우

1. CPU의 모든 레지스터를 TCB에 저장

2. 새 프로세스의 PCB에 저장된 페이지 테이블 주소를 CPU의 Page Table Base Register (PTBR)에 적재

- TLB 미스 시 페이지 테이블을 액세스하기 위해

3. TLB의 모든 항목 지우기

- 이전 프로세스의 매핑 정보가 들어 있기 때문, 그대로 두면 전혀 다른 물리 메모리로 매핑되는 문제 발생
- 큰 비용 대가
  - 새로운 프로세스의 실행이 시작되면 TLB 미스가 발생하고 TLB에 항목이 채워지기 시작함

4. 새로 스케줄된 스레드의 TCB에서 레지스터 값을 CPU에 적재

4. 페이지 테이블의 낭비 문제 해결

페이지 테이블의 메모리 낭비

- 32비트 CPU 환경에서 프로세스당 페이지 테이블 크기
  - 프로세스의 주소 공간
    - 4GB/4KB = $2^{32}/2^{12}$ = 2^{20} = 약 100만개의 페이지로 구성
  - **프로세스당 페이지 테이블의 크기**
    - 한 항목이 4바이트이면 2^{20}개 x 4바이트 = **4MB**
- 10MB의 메모리를 사용하는 프로세스가 있다고 하면
  - 실제 활용되는 페이지 테이블 항목 수
    - 10MB/4KB = $10x2^{20}/2^{12}$ = $10x2^{8}$ = 2560개
  - 페이지 테이블 항목 사용률
    - $10x2^{8}/2^{20}$ = $10/2^{12}$ = 0.0024
    - 매우 낮음

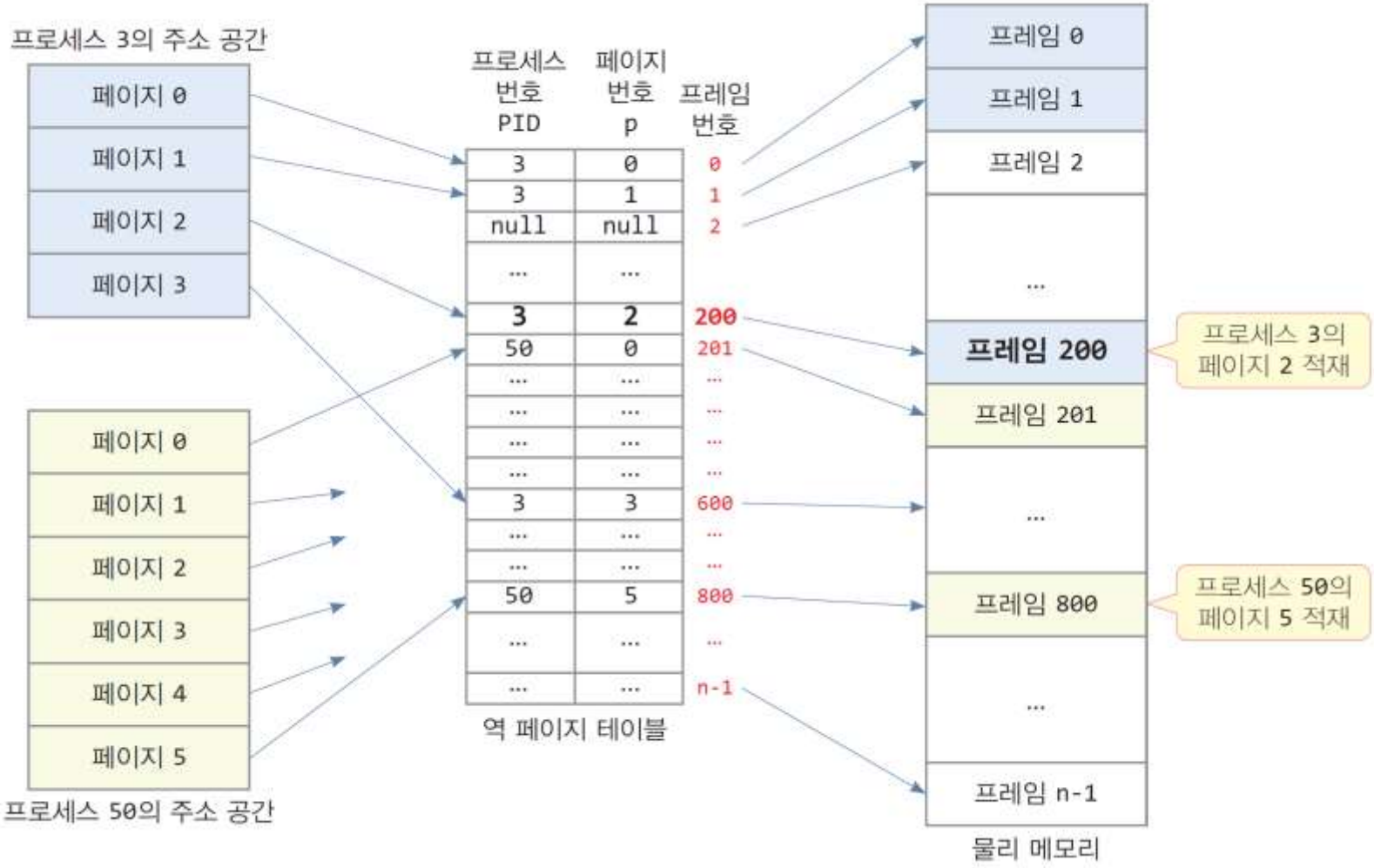
페이지 테이블 낭비 문제의 해결책

1. 역 페이지 테이블(inverted page table, IPT)
2. 멀티 레벨 페이지 테이블(multi-level page table) 오늘날

역 페이지 테이블

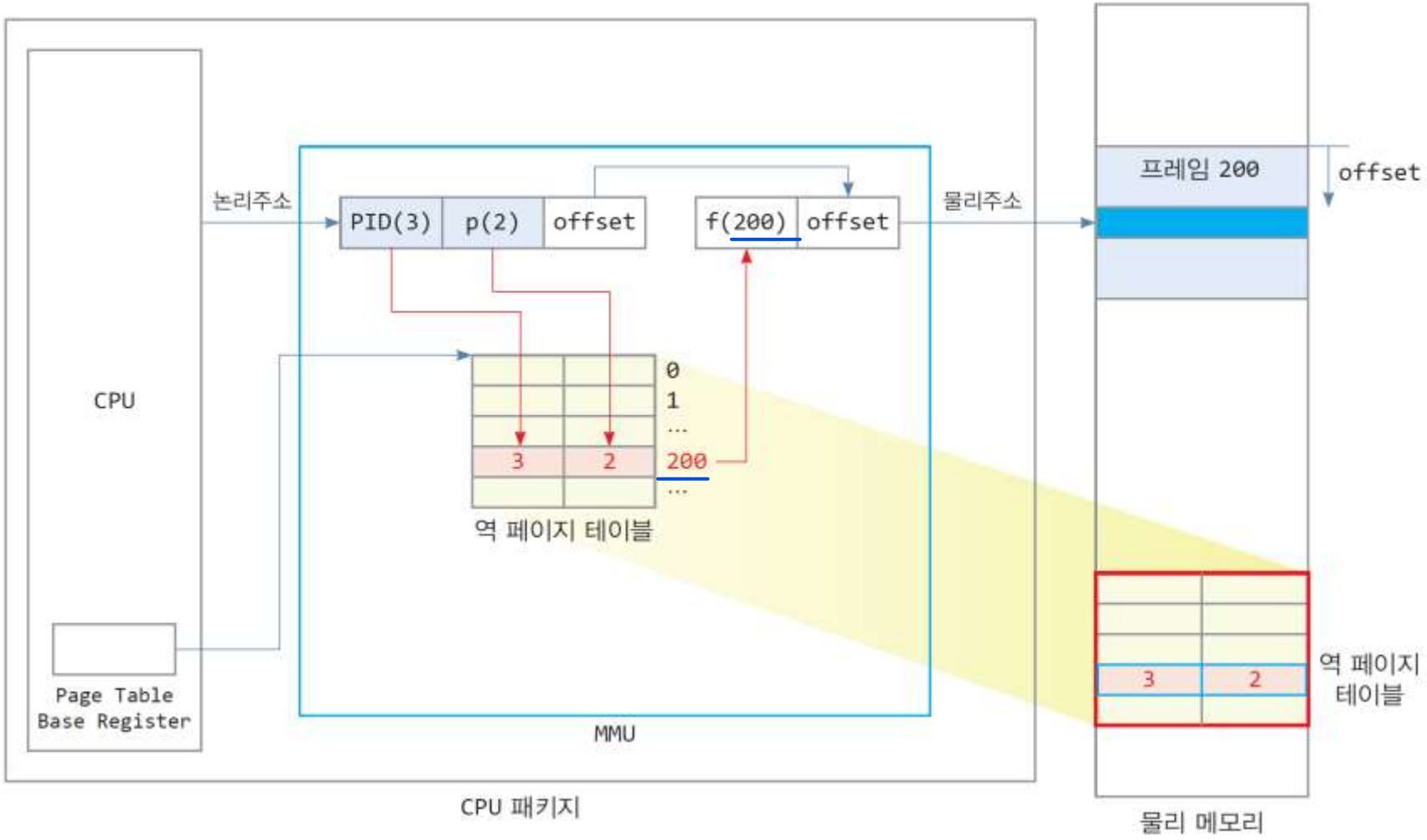
- 물리 메모리 전체 프레임에 대해, 각 프레임이 어떤 프로세스의 어떤 페이지에 할당되었는지를 나타내는 테이블
- 시스템에 1개의 역 페이지 테이블 둠
  - 역 페이지 테이블 항목의 수 = 물리 메모리의 프레임 개수
- 역 페이지 테이블 항목
  - [프로세스번호(pid), 페이지 번호(p)]
- 역 페이지 테이블의 인덱스
  - 프레임 번호
- 역 페이지 테이블을 사용할 때 논리 주소 형식
  - [프로세스번호, 페이지 번호, 옵셋]
- 역 페이지 테이블을 사용한 주소 변환
  - 논리 주소에서 (프로세스번호, 페이지 번호)로 역 페이지 테이블 검색
  - 일치하는 항목을 발견하면 항목 번호가 바로 프레임 번호임
  - 프레임 번호와 옵셋을 연결하면 물리 주소

역 페이지 테이블을 이용할 때 페이지와 프레임의 관계



역 페이지 테이블을 사용한 논리 주소의 물리 주소 변환

논리 주소 : [프로세스 번호, 페이지 번호, 옵셋]



역 페이지 테이블 크기

- 역 페이지 테이블 개수
  - 시스템에 1개 존재
- 역 페이지 테이블 크기
  - 역 페이지 테이블 항목 : 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 = 4GB/4KB = 2^{20}개 = 약 100만개
  - 역 페이지 테이블 크기는 컴퓨터에 설치된 물리 메모리 크기에 따라 달라짐
    - 예) 물리 메모리가 4GB, 프레임 크기가 4KB, 한 항목 크기가 8바이트라면
      역 페이지 테이블의 크기 = 2^{20}개 항목 x 8바이트 = 8MB
- 기존 페이지 테이블과 비교
  - 예) 10개의 프로세스가 실행 중일 때(물리 메모리가 4GB인 경우)
    기존 페이지 테이블 = 4MBx10개 = 40MB 크기
    역 페이지 테이블 = 8MB(기존의 1/5 수준)

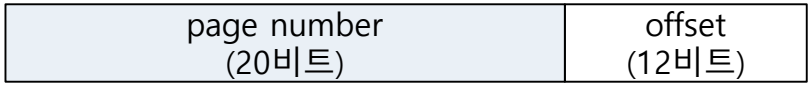
멀티레벨 페이지 테이블

- 멀티레벨 페이지 테이블 개념
  - 프로세스가 현재 사용 중인 페이지들에 대해서만 페이지 테이블을 만드는 방식
    - 기존 페이지 테이블의 낭비를 줄임
  - 페이지 테이블을 수십~수백 개의 작은 페이지 테이블로 나누고 이들을 여러 레벨(level)로 구성
  - 오늘날 대부분 운영체제에서 사용되고 있음

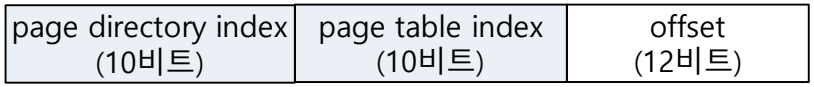
- 2-레벨로 멀티레벨 페이지 테이블을 구성하는 경우
  - 논리 주소 구성
    - [페이지 디렉터리 인덱스, 페이지 테이블 인덱스, 옵셋]
    - 페이지 크기 4KB
    - 논리 주소의 하위 12비트 : 페이지 내 옵셋 주소
    - 논리 주소의 상위 20비트 : 페이지 디렉터리 인덱스와 페이지 테이블 인덱스

2-레벨 페이지 테이블

- 페이지 테이블들의 페이지 화
  - 논리 주소는 페이지 번호와 옵셋으로 구성

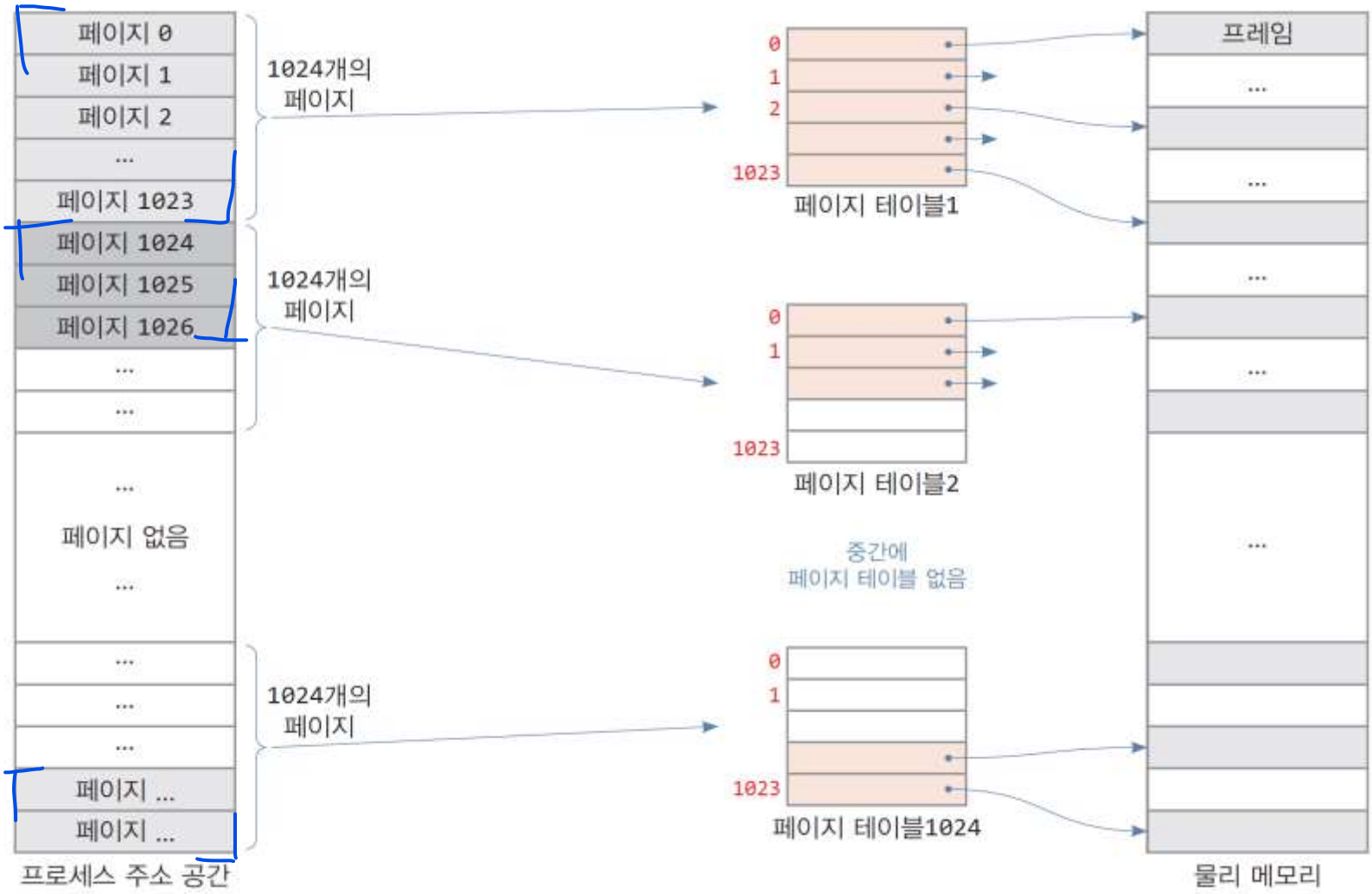


  - 페이지 번호 부분을 2개의 레벨로 나눔



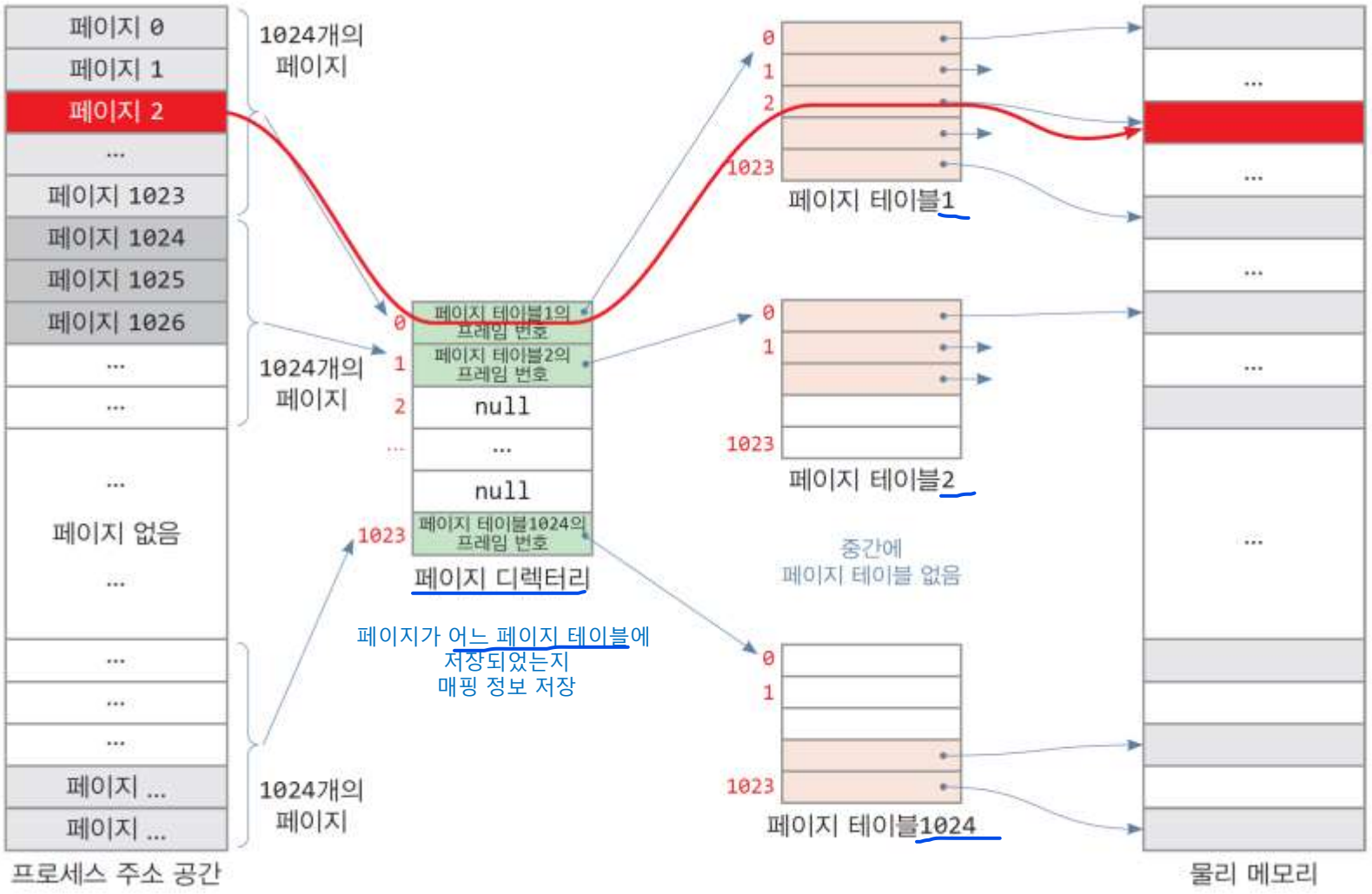
    - 페이지 디렉터리와 페이지 테이블의 트리 구조
    - 사용 중인 페이지들에 대해서만 페이지 테이블 할당

1024개의 페이지마다 1개의 페이지 테이블 사용

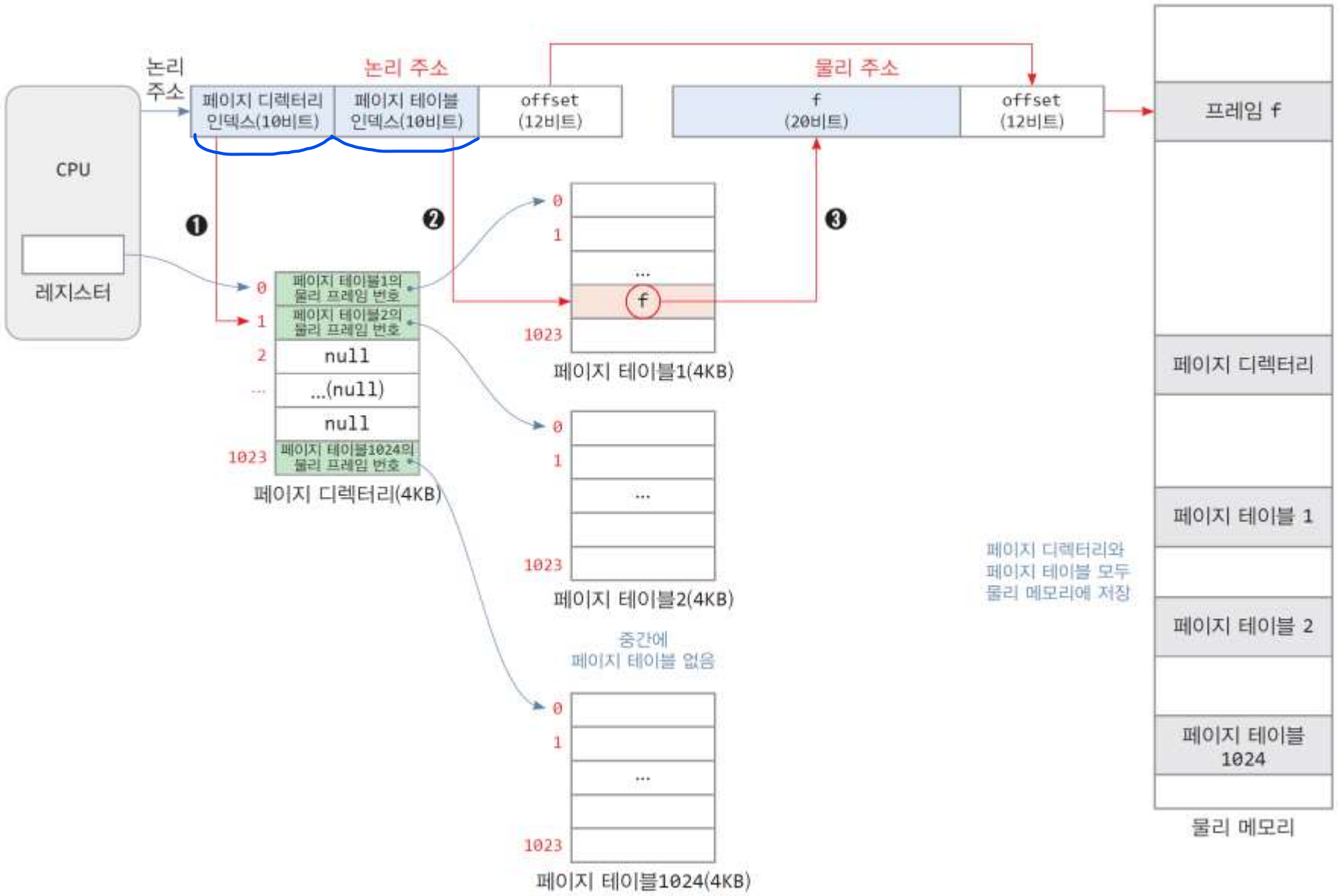


* 프로세스에 현재 할당된 페이지들에 대해서만 페이지 테이블 생성

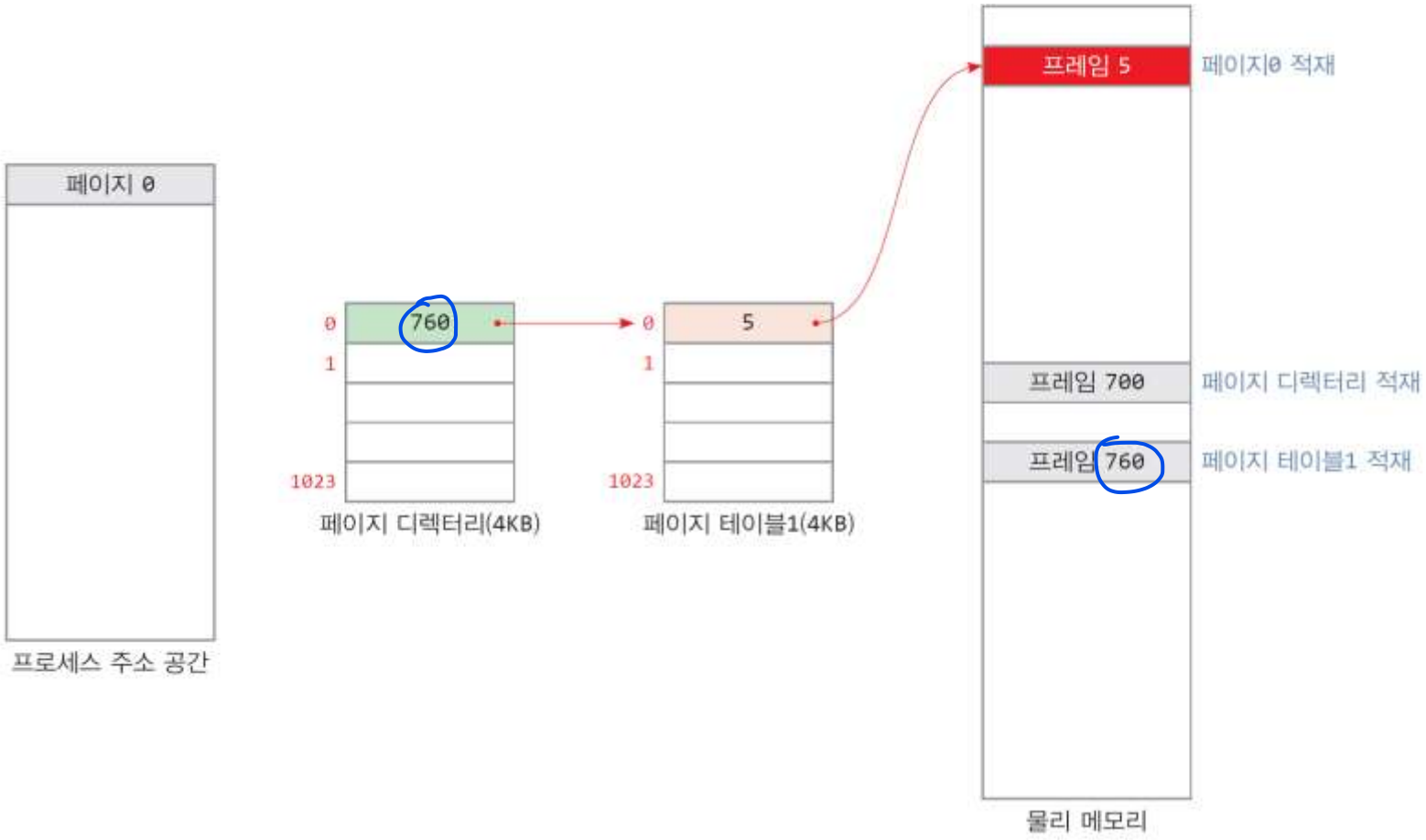
2-레벨 페이지 테이블 : 페이지 디렉터리와 페이지 테이블로 구성



2-레벨 페이지 테이블 사례

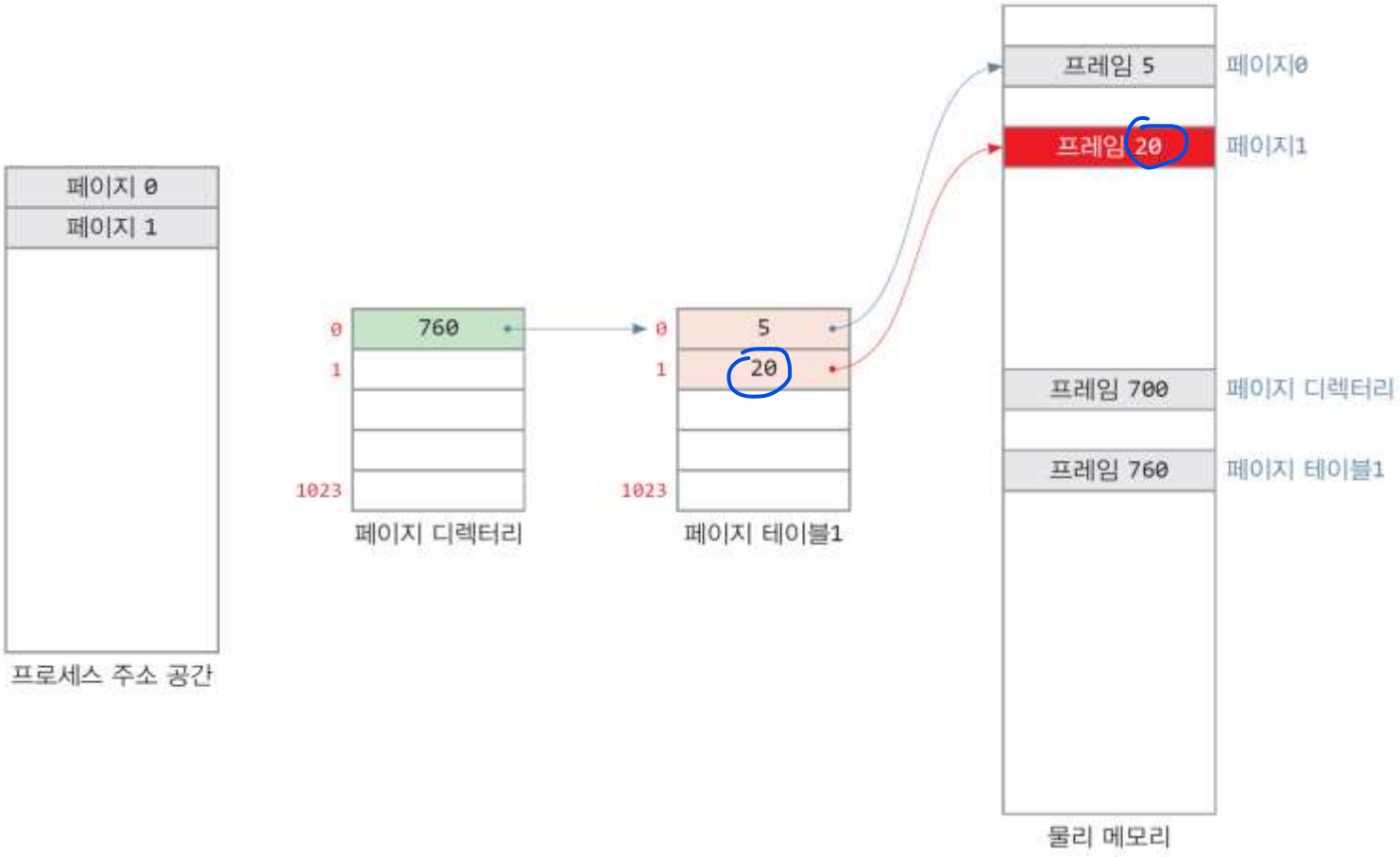


2-레벨 페이지 테이블이 형성되는 과정(1)



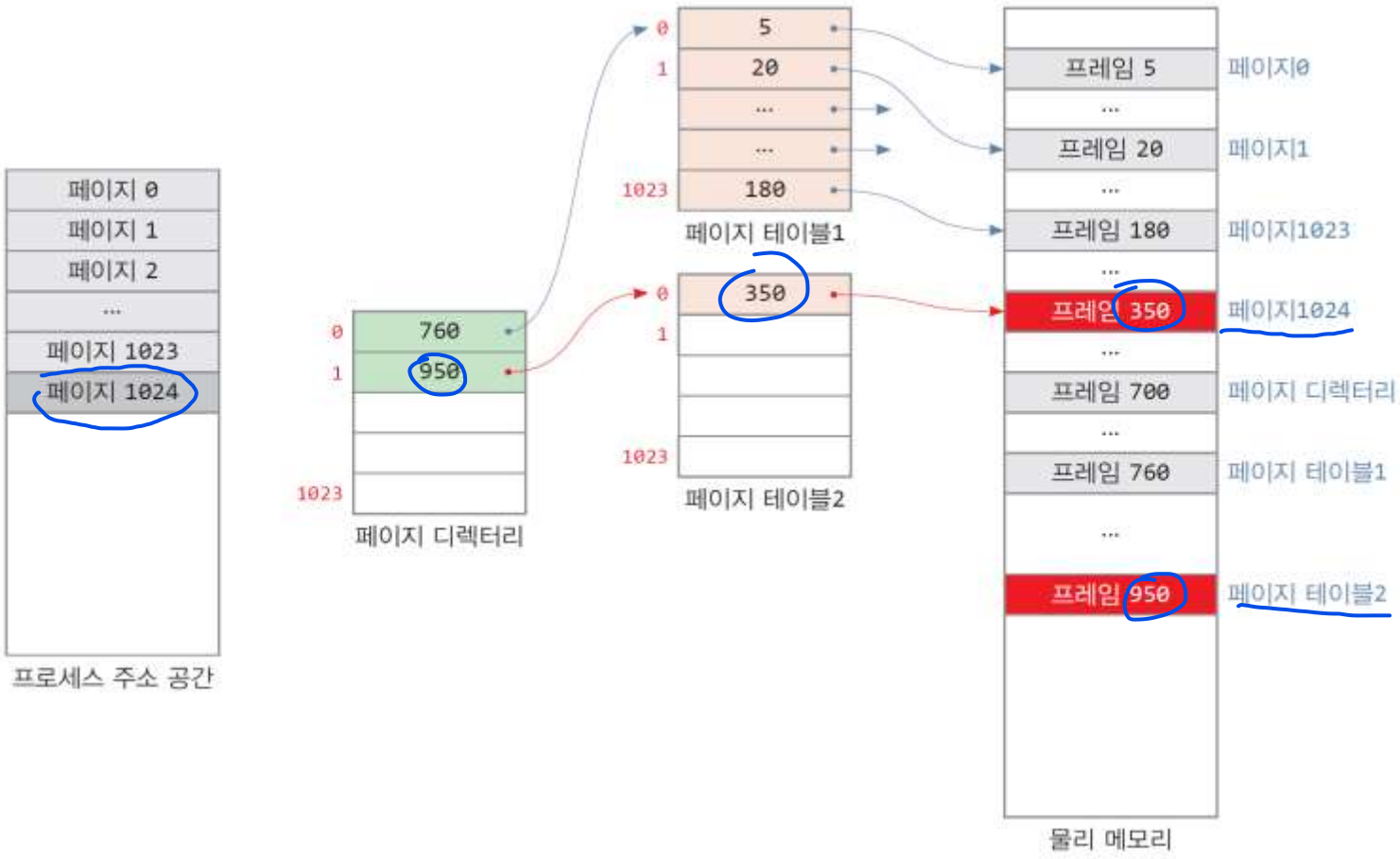
(1) 프로세스의 페이지 0이 프레임 5에 적재될 때

2-레벨 페이지 테이블이 형성되는 과정(2)



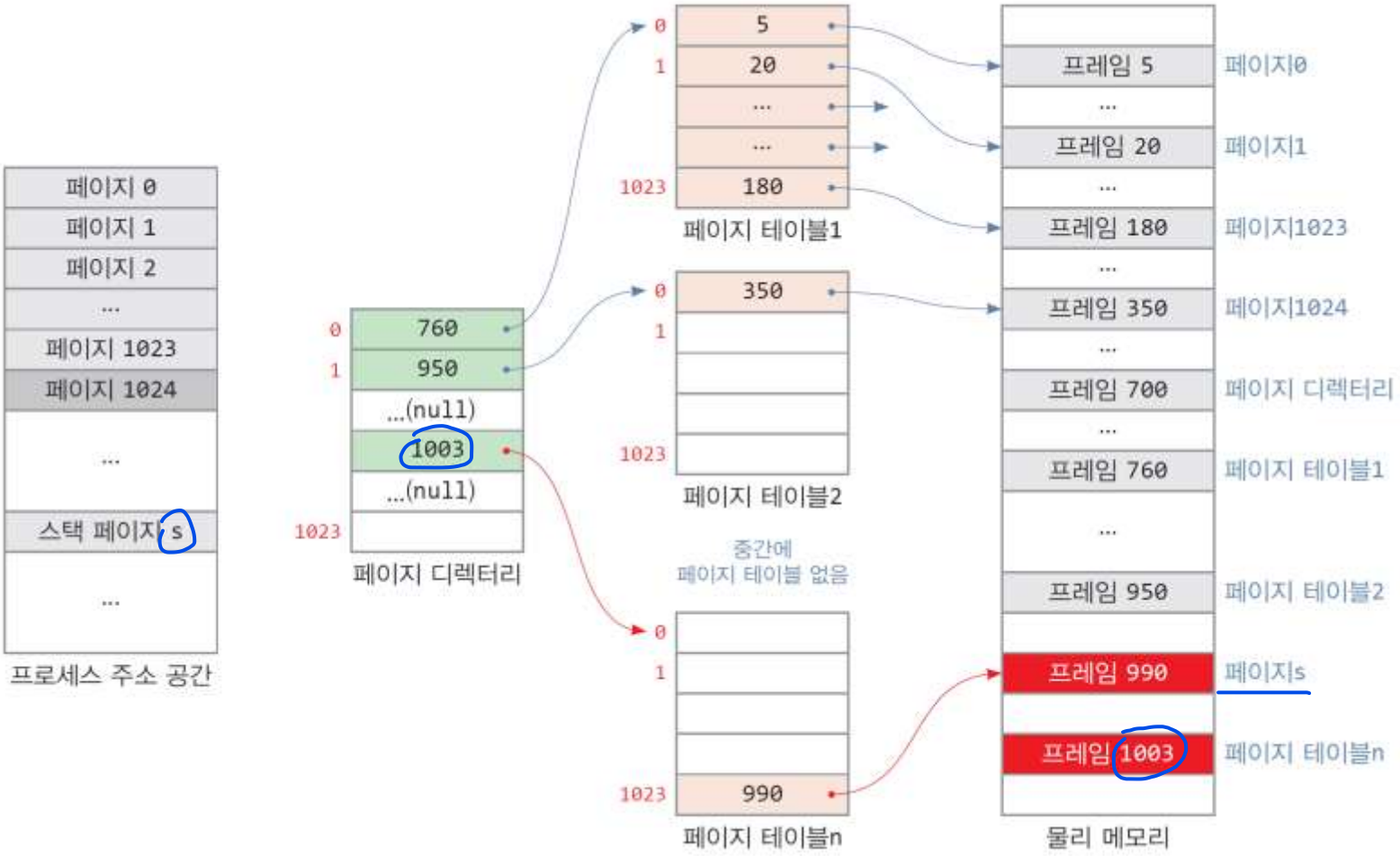
(2) 프로세스의 페이지 1이 프레임 20에 적재될 때

2-레벨 페이지 테이블이 형성되는 과정(3)



(3) 프로세스의 페이지 1024가 프레임 350에 적재될 때 – 페이지 테이블2 추가 생성

2-레벨 페이지 테이블이 형성되는 과정(4) – 교재 오류 수정



(4) 프로세스가 스택 영역에서 페이지(페이지 s, 프레임 990)를 할당받을 때

- 페이지 테이블 1024 추가 생성

이 사례에서 페이지 테이블의 크기
= 페이지 디렉터리 1개(4KB) + 3개의 페이지 테이블(12KB)
= 16KB

페이지 테이블의 크기가 대폭 감소됨

4MB → 16KB

2-레벨 페이지 테이블의 크기

- 2-레벨 페이지 테이블의 최대 메모리 소모량
  - 페이지 디렉터리 1개 + 최대 1024개의 페이지 테이블
  - = 4KB + 1024*4KB = 4KB + 4MB
  - 하지만, 대부분의 경우, 프로세스는 그리 크지 않음
- 사례 1 – 프로세스가 1000개의 페이지로 구성되는 경우(4MB의 프로세스)
  - 1000개의 페이지는 1개(4KB)의 페이지 테이블에 의해 매핑 가능
  - 메모리 소모량
    - 1개의 페이지 디렉터리와 1개의 페이지 테이블
    - = 4KB + 4KB = 8KB
- 사례 2 – 프로세스가 400MB 크기인 경우
  - 프로세스의 페이지 개수 = 400MB/4KB = 100×1024 개
    - 100개의 페이지 테이블 필요
  - 메모리 소모량
    - 1개의 페이지 디렉터리와 100개의 페이지 테이블
    - = 4KB + 100×4KB = 404KB
- 결론
  - 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.