

9장. 메인 메모리

순천향대학교 컴퓨터공학과 이 상 정

순천향대학교 컴퓨터공학과

1

운영체제

강의 목표 및 내용

□ 목표

- 메모리 하드웨어를 구성하는 다양한 방법을 기술
- 페이징과 세그멘테이션 기법을 포함한 다양한 메모리 관리 기법들을 설명

□ 내용

- 배경
- 스와핑
- 연속 메모리 할당
- 세그멘테이션
- 페이징
- 페이지 테이블 구조
- Intel 32, 64-비트 구조와 ARM 사례

순천향대학교 컴퓨터공학과

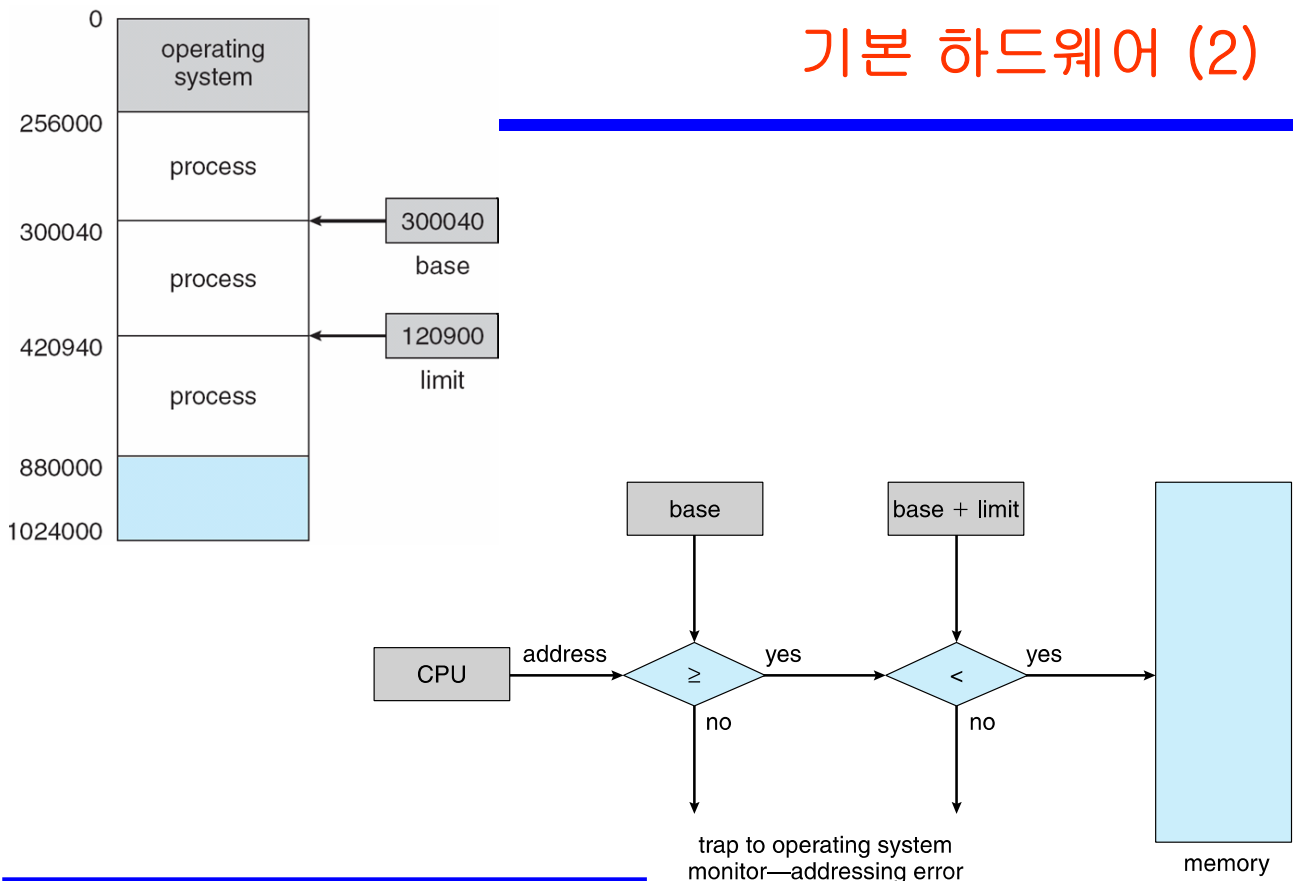
2

9. 메인 메모리

기본 하드웨어 (1)

- 각각의 프로세스는 독립된 메모리 공간을 가짐
 - 특정 프로세스만 접근할 수 있는 합법적인(legal) 메모리 주소 영역을 설정
 - 프로세스가 합법적인 영역만을 접근하도록 보호하는 것이 필요
- 메모리 공간의 보호는 CPU 하드웨어가 사용자 모드에서 만들어진 모든 주소와 레지스터를 비교함으로써 이루어짐
 - 베이스(base) 레지스터와 상한(limit) 레지스터 사용
 - 베이스 레지스터는 가장 작은 합법적인 물리 메모리 주소의 값을 저장
 - 상한 레지스터는 주어진 영역의 크기를 저장

기본 하드웨어 (2)



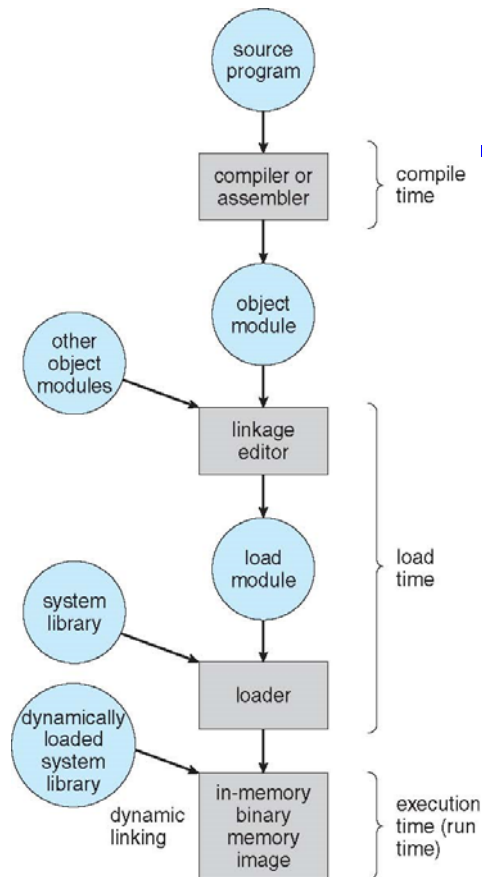
주소의 할당 (Address Binding)

- 프로그램이 실행되기 위해서는 메인 메모리에 적재되어 프로세스가 되어야 함
 - 프로그램은 원래 이진 실행 파일 형태로 디스크에 저장
- 입력 큐(input queue)
 - 디스크에서 메인 메모리로 적재되기를 기다리고 있는 프로세스들의 집합
 - 큐에서 하나의 프로세스를 선택해서, 메모리에 적재 후 실행하고, 프로세스가 실행되는 동안 메모리에서 명령어와 자료를 액세스
- 주소는 각 단계에서 다양하게 표현
 - 소스 코드에서는 **심볼(변수)**로 표시
 - 컴파일된 코드에서는 **재배치 가능한 주소(relocatable address)**로 바인딩
 - 예, 모듈의 시작에서부터 14바이트
 - 링커나 로더는 재배치 가능한 주소를 **절대주소**로 바인딩
 - 각 바인딩 과정은 한 주소 공간에서 다른 주소 공간으로 맵핑

메모리 주소 공간에서 명령어와 자료의 바인딩

- 메모리 주소 공간에서 **명령어와 자료의 바인딩**은 이루어지는 시점에 따라 다음과 같이 구분
 - **컴파일 시간(Compile time) 바인딩**
 - 만일 프로세스가 메모리 내에 들어갈 위치를 컴파일 시간에 미리 알 수 있으면 컴파일러는 **절대 코드**를 생성할 수 있음
 - 만일 이 위치가 변경되어야 한다면 이 코드는 다시 컴파일 되어야 함
 - MS-DOS의 .COM 양식 프로그램
 - **적재 시간(Load time) 바인딩**
 - 만일 프로세스가 메모리 위치를 컴파일 시점에 알지 못하면 컴파일러는 일단 이진 코드를 **재배치 가능 코드(relocatable code)**로 만들어야 함
 - **실행 시간(Execution time) 바인딩**
 - 만약 프로세스가 **실행하는 중간**에 메모리 내의 한 세그먼트로부터 다른 세그먼트로 옮겨질 수 있다면 바인딩이 실행 시간까지 지연
 - 특별한 하드웨어 지원이 필요
 - 베이스(base) 레지스터와 상한(limit) 레지스터사용

사용자 프로그램의 단계별 처리 과정



보충 학습: 링커 (linker)

- 서로 독자적으로 생성된 목적파일들을 하나로 연결하여 실행 이미지(executable image, 실행파일) 생성
 - 링크 에디터(link editor) 또는 링커라 한다.
- 실행 이미지 생성 과정
 1. 세그먼트를 통합
 2. 레이블들의 주소를 결정
 3. 외부 및 내부 참조 해결

보충 학습: 목적파일의 링크 예 (1)

| Object file header | | | |
|------------------------|-----------|--------------------|------------|
| | Name | Procedure A | |
| | Text size | 100 _{hex} | |
| | Data size | 20 _{hex} | |
| Text segment | Address | Instruction | |
| | 0 | lw \$a0, 0(\$gp) | |
| | 4 | jal 0 | |
| Data segment | ... | ... | |
| | 0 | (X) | |
| | ... | ... | |
| Relocation information | Address | Instruction type | Dependency |
| | 0 | lw | X |
| | 4 | jal | B |
| Symbol table | Label | Address | |
| | X | - | |
| | B | - | |

| Object file header | | | |
|------------------------|-----------|--------------------|------------|
| | Name | Procedure B | |
| | Text size | 200 _{hex} | |
| | Data size | 30 _{hex} | |
| Text segment | Address | Instruction | |
| | 0 | sw \$a1, 0(\$gp) | |
| | 4 | jal 0 | |
| Data segment | ... | ... | |
| | 0 | (Y) | |
| | ... | ... | |
| Relocation information | Address | Instruction type | Dependency |
| | 0 | sw | Y |
| | 4 | jal | A |
| Symbol table | Label | Address | |
| | Y | - | |
| | A | - | |

□ 외부 참조

- 프로시저 A, B
- 데이터워드 X, Y

□ 목적 파일 1

- 프로시저 A 정의
- X 정의, 참조
- B 호출

□ 목적 파일 2

- 프로시저 B 정의
- Y 정의, 참조
- A 호출

9. 메인 메모리

운영체제

컴퓨터 구조 복습: 목적파일의 링크 예 (2)

| Executable file header | | |
|------------------------|--------------------------|-------------------------------------|
| | Text size | 300 _{hex} |
| | Data size | 50 _{hex} |
| Text segment | Address | Instruction |
| | 0040 0000 _{hex} | lw \$a0, 8000 _{hex} (\$gp) |
| | 0040 0004 _{hex} | jal 40 0100 _{hex} |
| | ... | ... |
| | 0040 0100 _{hex} | sw \$a1, 8020 _{hex} (\$gp) |
| | 0040 0104 _{hex} | jal 40 0000 _{hex} |
| Data segment | ... | ... |
| | Address | |
| | 1000 0000 _{hex} | (X) |
| | ... | ... |
| | 1000 0020 _{hex} | (Y) |
| | ... | ... |

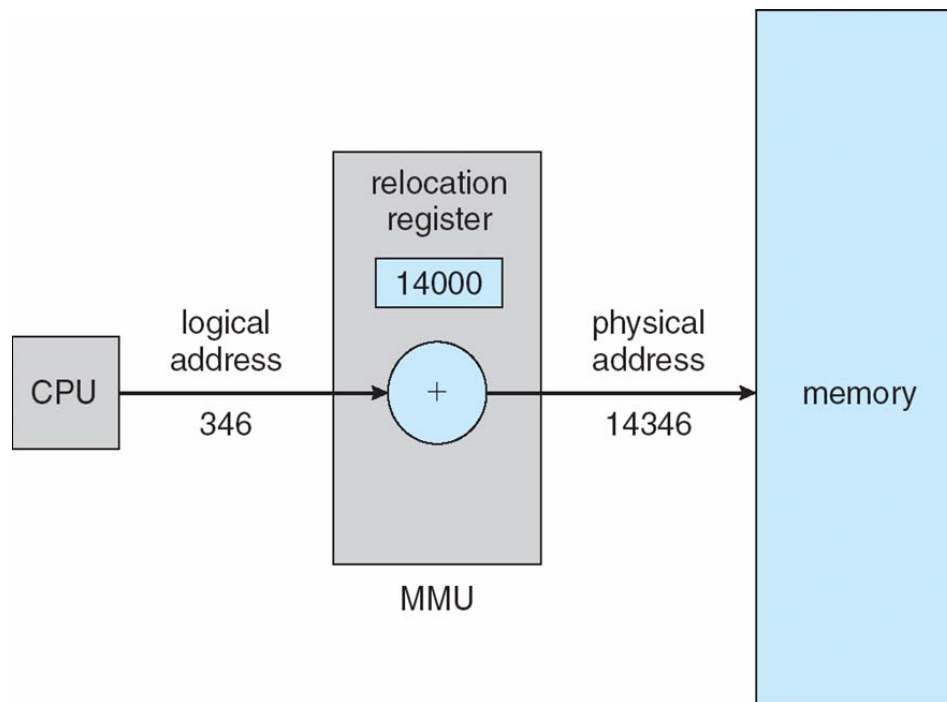
논리 대 물리 주소 공간 (Logical Versus Physical Address Space)

- 논리 주소(logical address)
 - CPU(프로그램)가 생성하는 주소
- 물리주소(physical address)
 - 메모리가 취급하게 되는 주소
(즉, 메모리 주소 레지스터(MAR)에 주어지는 주소)
- 컴파일 시 바인딩과 적재 시의 바인딩 기법
 - 논리, 물리 주소가 같음
- 실행 시간 바인딩 기법
 - 논리, 물리 주소가 다름
 - 이 경우 논리 주소를 가상주소(virtual address)라 함

메모리 관리 장치 (MMU, Memory Management Unit)

- 가상 주소를 물리 주소로 변환(mapping) 하는 하드웨어 장치
 - MMU 방식에서는 사용자 프로세스에 의해 생성된 모든 주소에 재배치(relocation) 레지스터의 값이 더해짐
 - 재배치 레지스터는 앞에서 기술한 일종의 기준 레지스터(base register)
- 사용자 프로그램은 논리 주소만을 다루고 실제적인 물리 주소를 결코 알 수 없음
 - 메모리 참조할 때 실행 시간 바인딩이 발생
 - 논리 주소가 물리 주소 변환

재배치 레지스터를 이용한 동적 재배치



동적 적재 (Dynamic Loading)

- 동적 적재에서 각 루틴은 **실제 호출되기** 전까지는 메모리에 올라오지 않고 재배치 가능한 상태로 디스크에서 대기
 - 향상된 메모리 공간 활용
 - 사용되지 않는 루틴은 메모리에 적재되지 않음
 - 오류 처리 루틴과 같이 아주 간혹 발생하면서도 많은 양의 코드를 필요로 하는 경우에 특히 유용
- 동적 적재는 운영체제로부터 특별한 지원을 필요로 하지 않음
 - 사용자 자신이 프로그램의 설계를 책임
 - 운영체제는 동적 적재를 구현하는 **라이브러리 루틴**을 제공 가능

동적 연결 및 공유 라이브러리 (Dynamic Linking & Shared Library)

정적 연결(static linking)

- 로더에 의해 시스템 라이브러리와 프로그램 코드가 이진 프로그램 이미지로 결합

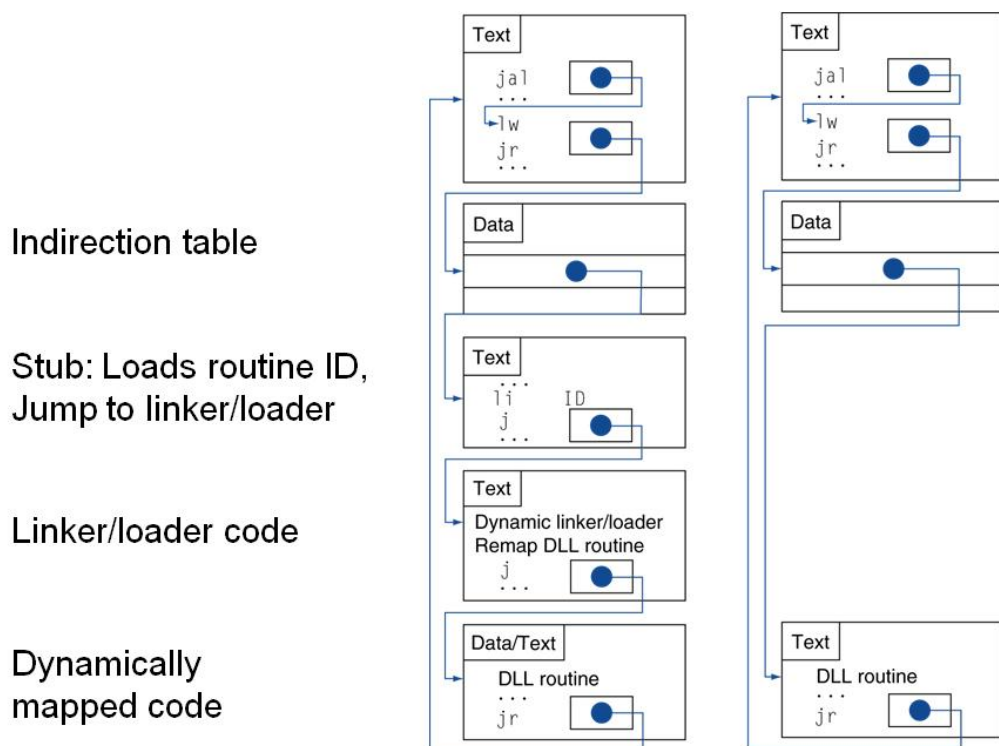
동적 연결에서는 연결(linking)이 실행시기까지 연기

- 동적 연결은 주로 공유 라이브러리(shared library)에 사용
- 동적 연결을 지원하지 않으면 모든 시스템 라이브러리를 부르는 프로그램들은 그들의 이진 프로그램 이미지 내에 시스템 라이브러리 루틴들을 한 부씩 가지고 있어야 함

작은 코드 조각인 스텝(stub)을 사용하여 메모리에 적재된 라이브러리의 위치를 찾음

- 라이브러리가 메모리에 없으면 디스크에서 가져옴
- 스텝 자신을 찾은 루틴의 번지로 대체하고, 실행

보충 학습: 지연 링키지 (Lazy Linkage)



스와핑 (Swapping)

□ 스와핑

- 프로세스는 실행도중에 임시로 보조 메모리(디스크)로 보내어졌다가 다시 메모리로 되돌아 올 수 있음

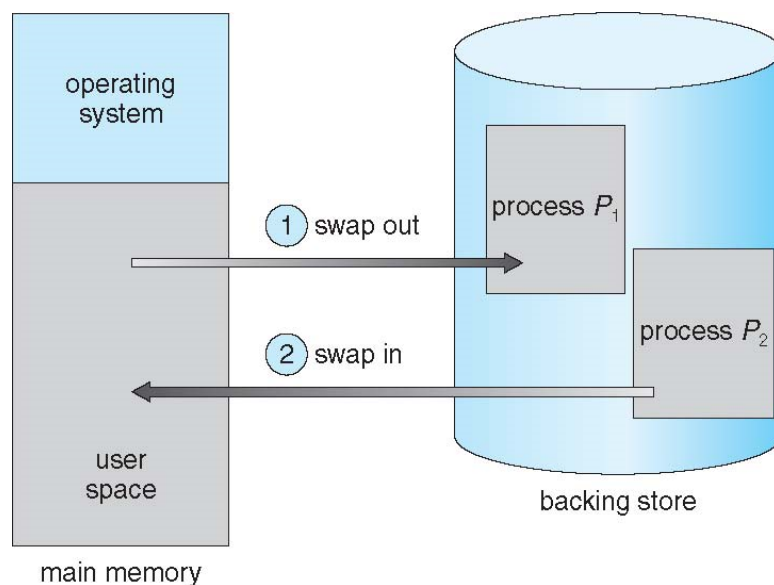
□ 스와핑은 우선순위 기반 스케줄링 알고리즘에 적용될 수 있음

- 낮은 우선순위 프로세스를 디스크로 스왑
- 높은 우선순위 프로세스가 끝나면, 낮은 우선순위 프로세스는 다시 메모리로 스왑
- 우선순위 기반 알고리즘에서 사용되는 스와핑의 변형을 롤 인(roll-in), 롤 아웃(roll-out)이라고 함

□ 스왑 시간의 대부분이 디스크 전송 시간

- 전송 시간은 스왑될 메모리의 크기와 비례

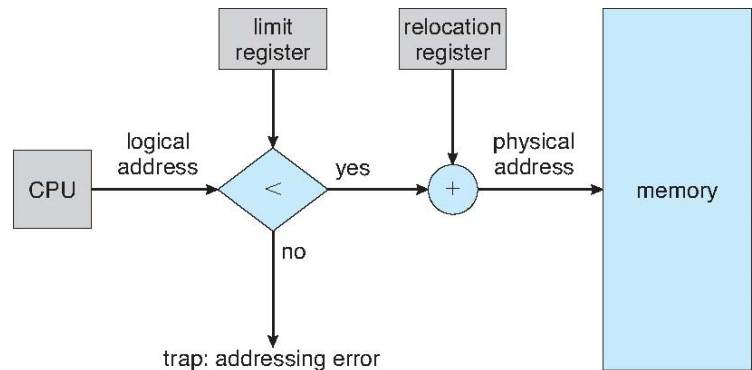
스와핑 예



연속 메모리 할당 (1) (Contiguous Memory Allocation)

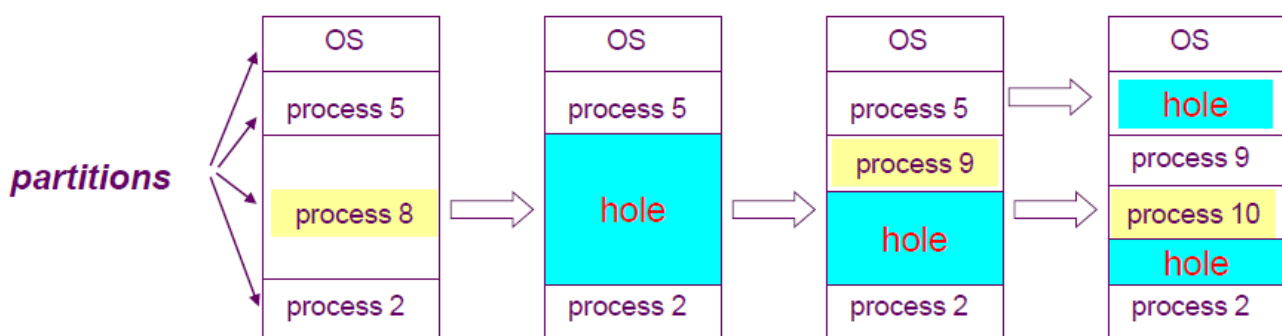
- 메모리는 일반적으로 두 개의 부분으로 분할
 - 메모리에 상주하는 **운영체제**로, 일반적으로 **인터럽트 벡터**와 함께 하위 메모리에 위치
 - 상위 메모리에 있는 **사용자 프로세스들**
- 연속 메모리 할당 시스템에서는 각 프로세스는 연속된 메모리 공간을 차지

- 메모리 사상과 보호
 - 재배치 레지스터와 상한 레지스터(limit register)에 의해 수행



연속 메모리 할당 (2)

- 메모리 할당 (memory allocation)
 - 고정된 크기로 분할(single partition)
 - 서로 다른 크기로 분할(**가변 분할**, multiple partition)
 - 사용 가능한 메모리 블록인 **공간(hole)**의 크기가 다양하며 메모리에 분산
 - 새 프로세스가 도착하면 이를 수용할 수 있는 충분한 크기의 공간을 할당
 - 운영체제는 할당된 분할과 가용한 분할(hole)에 관한 정보를 유지해야 함



동적 메모리 할당 문제 (Dynamic Storage Allocation Problem)

- 일련의 공간들-리스트로부터 크기 n -바이트 블록을 요구하는 것을 어떻게 만족시켜 줄 것이냐를 결정하는 문제
 - 최초 적합 (first-fit)
 - 첫 번째 사용 가능한 공간을 할당
 - 최적 적합 (best-fit)
 - 사용 가능한 공간들 중에서 가장 작은 것을 선택
 - 리스트가 정렬 되어 있지 않다면 전 리스트를 검색
 - 많은 작은 공간들이 생성
 - 최악 적합 (worst-fit)
 - 가장 큰 공간을 선택
 - 리스트가 정렬 되어 있지 않다면 전 리스트를 검색
 - 할당해 주고 남게 되는 공간은 충분히 커서 다른 프로세스들을 위하여 유용하게 사용
 - 최초 적합과 최적 적합 모두가 시간과 메모리 이용 효율 측면에서 최악 적합보다 낫다는 것이 입증

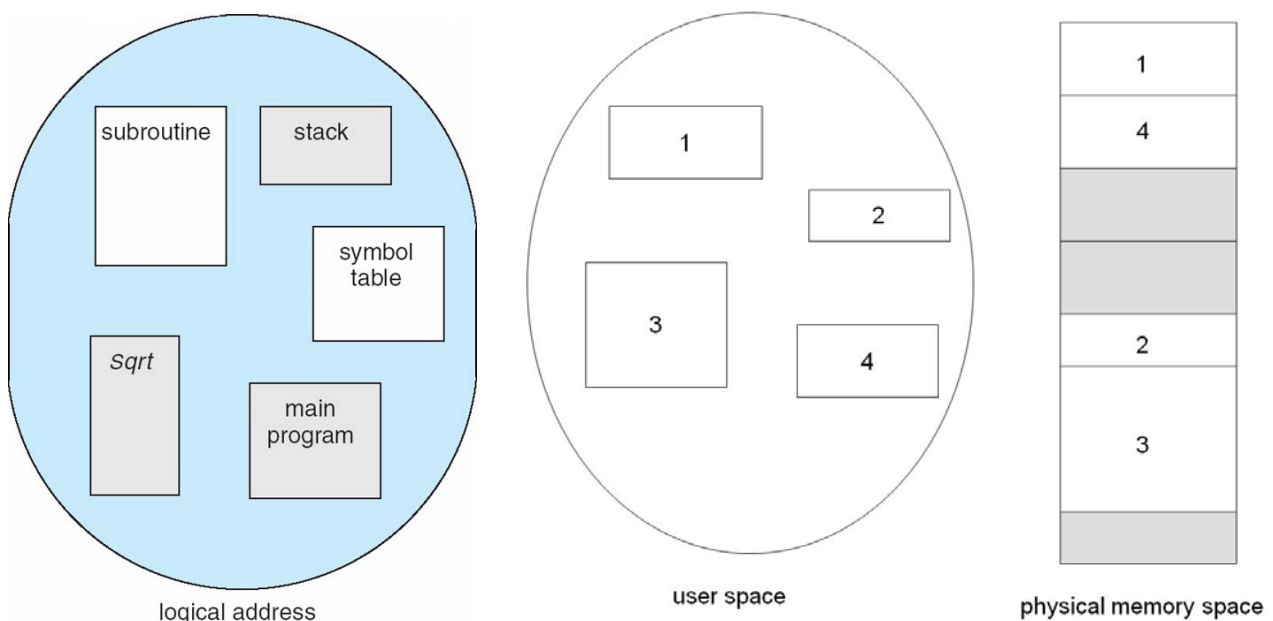
단편화 (Fragmentation)

- 단편화는 공간 중 일부가 사용 못하게 되는 것을 말함
 - 외부 단편화(external fragmentation)
 - 유휴 공간들을 모두 합치면 충분한 공간이 되지만 그것들이 너무 작은 조각들로 여러 곳에 분산되어 있을 때 발생
 - 내부 단편화(internal fragmentation)
 - 일반적으로 메모리는 고정된 크기의 정수 배로 할당되어 할당된 공간이 요구된 공간보다 약간 더 클 수 있음
- 외부 단편화는 압축(compaction)을 하여 해결
 - 메모리 모든 내용들을 한군데로 몰고 모든 자유 공간들을 다른 한 군데로 몰아서 큰 블록을 생성
 - 압축은 프로세스들의 재배치가 실행 시간에 동적으로 이루어지는 경우에만 가능

세그먼테이션 (Segmentation)

- 사용자 관점에서 지원되는 메모리 관리 방식
- 프로그램은 세그먼트의 집합
- 세그먼트는 다음과 같은 것들의 논리적 단위
 - 주 프로그램 (main())
 - 프로시저
 - 함수
 - 메서드
 - 객체
 - 광역변수, 지역 변수
 - 스택
 - 심볼 테이블, 배열

프로그램의 사용자 관점



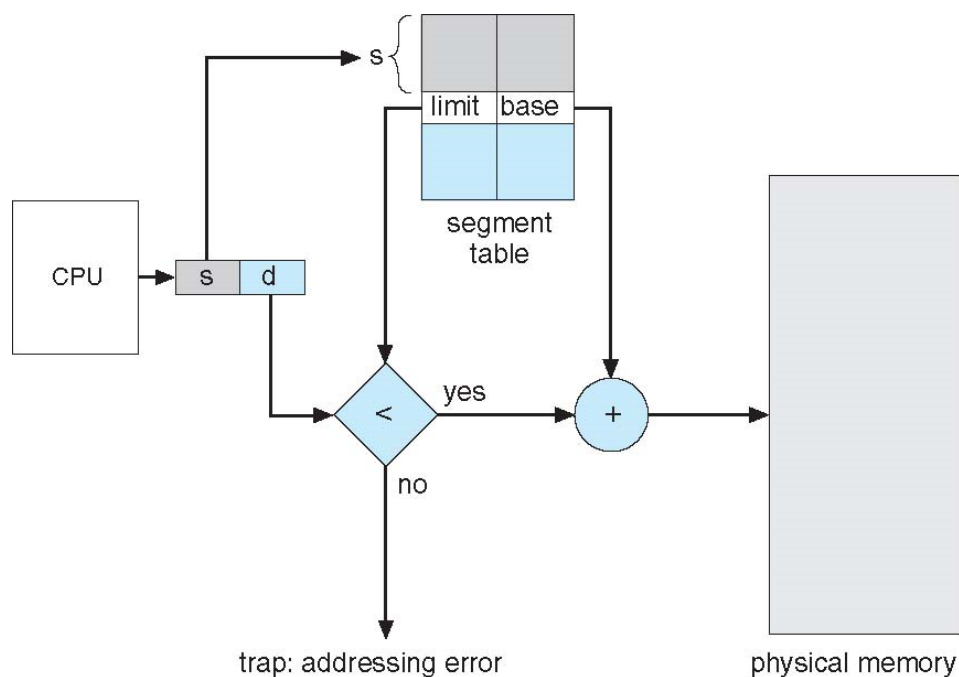
세그먼테이션 하드웨어 (1)

□ 논리 주소는 두 부분으로 구성
 <segment-number, offset>

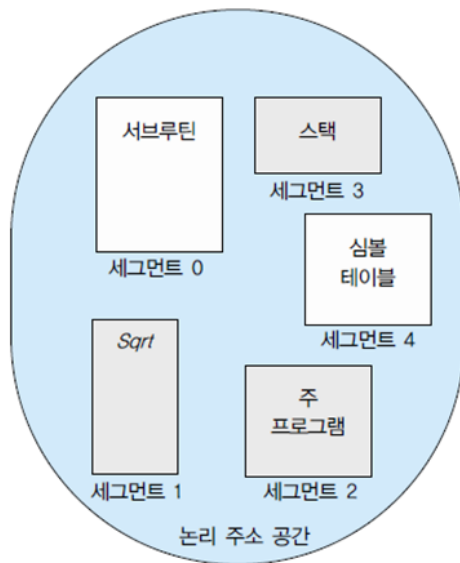
□ 세그먼트 테이블(segment table)

- 사용자가 정의한 이차원 주소는 일차원의 실제 주소로 사상
- 테이블의 각 항목
 - 세그먼트의 **기준 (base)**
 - 세그먼트의 시작 주소를 표시
 - 세그먼트 **한계(limit)**
 - 세그먼트의 길이를 명시

세그먼테이션 하드웨어 (2)

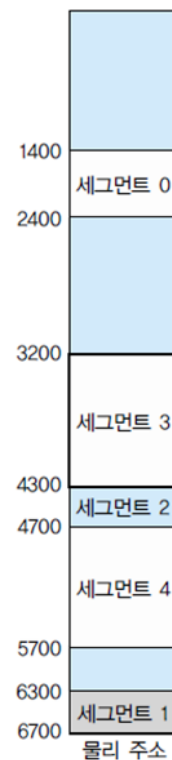


세그먼트 예



| | 한계 | 기준 |
|---|------|------|
| 0 | 1000 | 1400 |
| 1 | 400 | 6300 |
| 2 | 400 | 4300 |
| 3 | 1100 | 3200 |
| 4 | 1000 | 4700 |

세그먼트 테이블

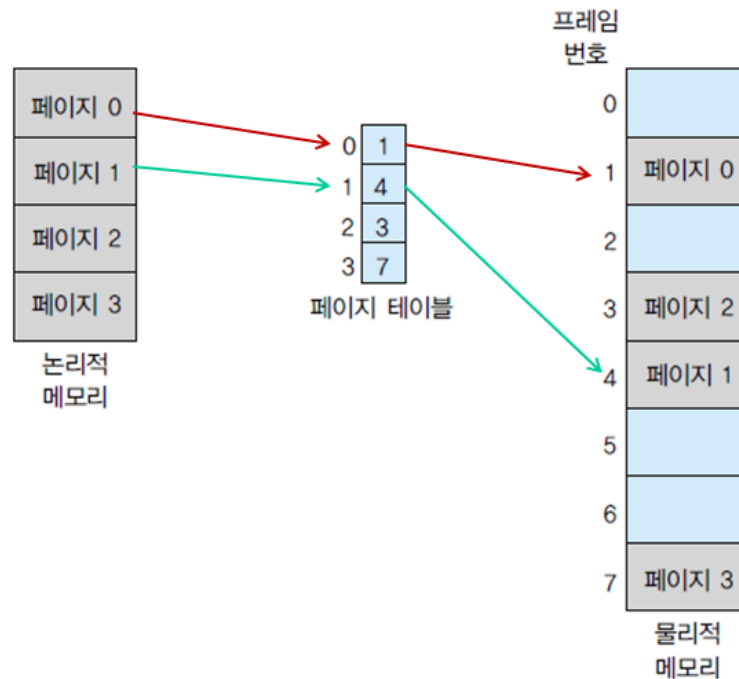


9. 메인 메모리

페이징 (Paging)

- ❑ 페이징은 논리 주소 공간이 한 연속적인 공간에 다 모여 있어야 한다는 제약을 제거
 - 프로세스는 물리 메모리에 빈 공간이 있으면 할당
- ❑ 기본 방법
 - 물리 메모리는 프레임(frame)이라 불리우는 같은 크기 블록으로 분할
 - 크기는 2의 멍승으로 512바이트에서 16M 바이트 범위
 - 논리 메모리는 페이지(page)라 불리는 같은 크기의 블록으로 분할
 - 한 프로세스가 실행될 때 프로세스의 페이지는 보조 메모리로부터 메인 메모리 프레임으로 들어 감
 - 모든 자유 프레임 리스트들은 추적 관리
 - 논리 주소에서 물리 주소로 변환하는 페이지 테이블(page table) 필요
 - 내부 단편화

페이징 예

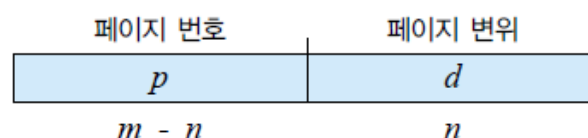


주소 변환 (Address Translation)

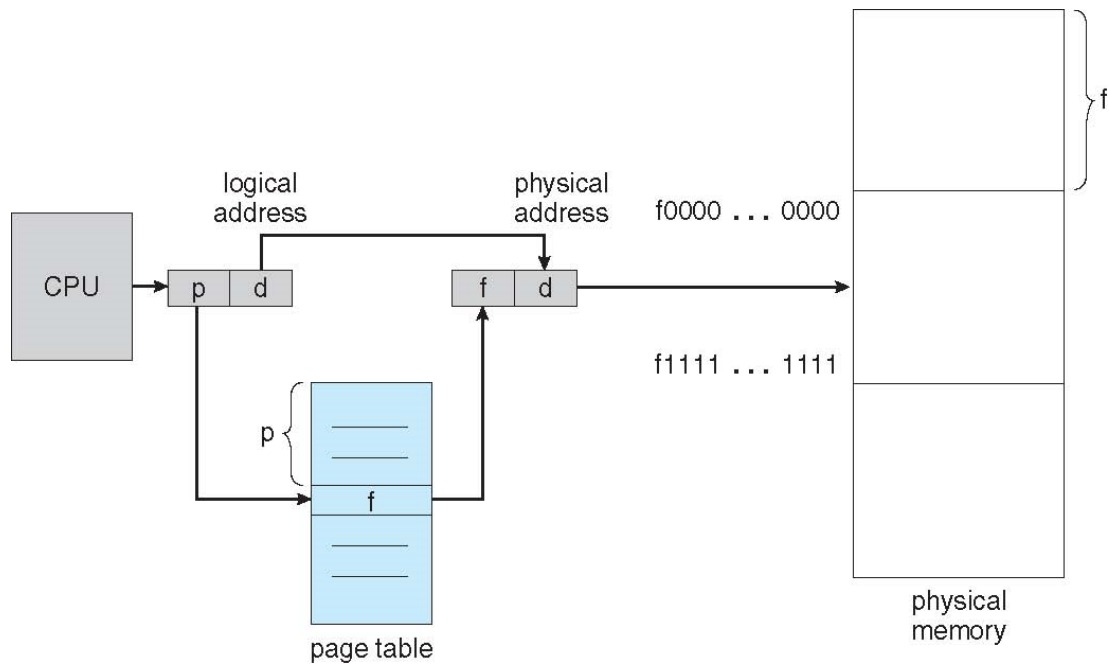
□ CPU에 의해 생성된 주소는 두 부분으로 분할

- **페이지 번호 (p)**
 - 페이지 테이블을 액세스할 때 사용
 - 페이지 테이블에는 물리 메모리에서의 베이스 주소(base address)가 저장
- **페이지 변위(d, offset)**
 - 페이지 테이블의 베이스 주소에 페이지 변위를 더하면 물리 주소가 생성
- 예

논리 주소 공간의 크기가 2^m 이고, 페이지가 2^n 크기라면 논리 주소의 상위 $m - n$ 비트는 페이지 번호를 나타내고, 하위 n 비트는 페이지 변위를 나타냄



페이징 하드웨어



운영체제

| | | |
|---|----|---|
| 0 | 0 | a |
| | 1 | b |
| | 2 | c |
| | 3 | d |
| 1 | 4 | e |
| | 5 | f |
| | 6 | g |
| | 7 | h |
| 2 | 8 | i |
| | 9 | j |
| | 10 | k |
| | 11 | l |
| 3 | 12 | m |
| | 13 | n |
| | 14 | o |
| | 15 | p |

논리 메모리

| | |
|---|---|
| 0 | 5 |
| 1 | 6 |
| 2 | 1 |
| 3 | 2 |

페이지 테이블

| 페이지 번호 | 페이지 변위 |
|--------|--------|
| p | d |

| | |
|----|------------------|
| 0 | |
| 4 | i j k l |
| 8 | m n o p |
| 12 | |
| 16 | |
| 20 | a b c d |
| 24 | e f g h |
| 28 | |

물리 메모리

주소변환 예

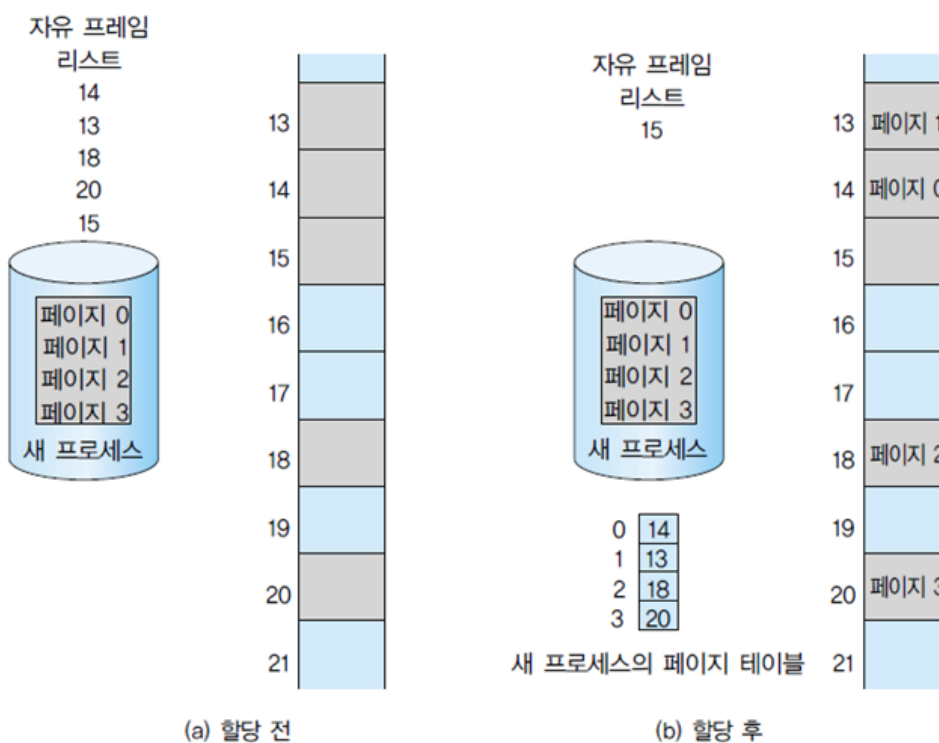
- n=2, m=4
- 32-바이트 메모리

페이징 - 내부 단편화

□ 내부 단편화 계산

- 페이지 크기 = 2048 바이트
- 프로세스 크기 = 72766 바이트
= 35 페이지 + 1086 바이트
 - 내부 단편화 = 2048 - 1086 = 962 바이트
- 최악의 경우 단편화 = 1 프레임 - 1 바이트
- 평균 단편화 = $\frac{1}{2}$ 프레임 크기
- 프레임 크기가 작아지면 단편화 감소?
 - 페이지 테이블 크기 증가
- 페이지 크기가 증가하는 추세
 - 보통 2 ~ 8 KB

자유 프레임 (Free Frame)

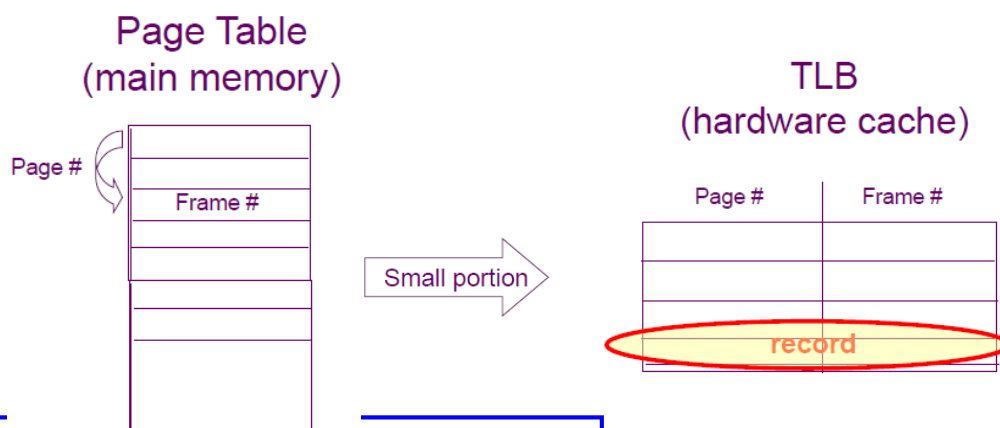


페이지 테이블 하드웨어 구현

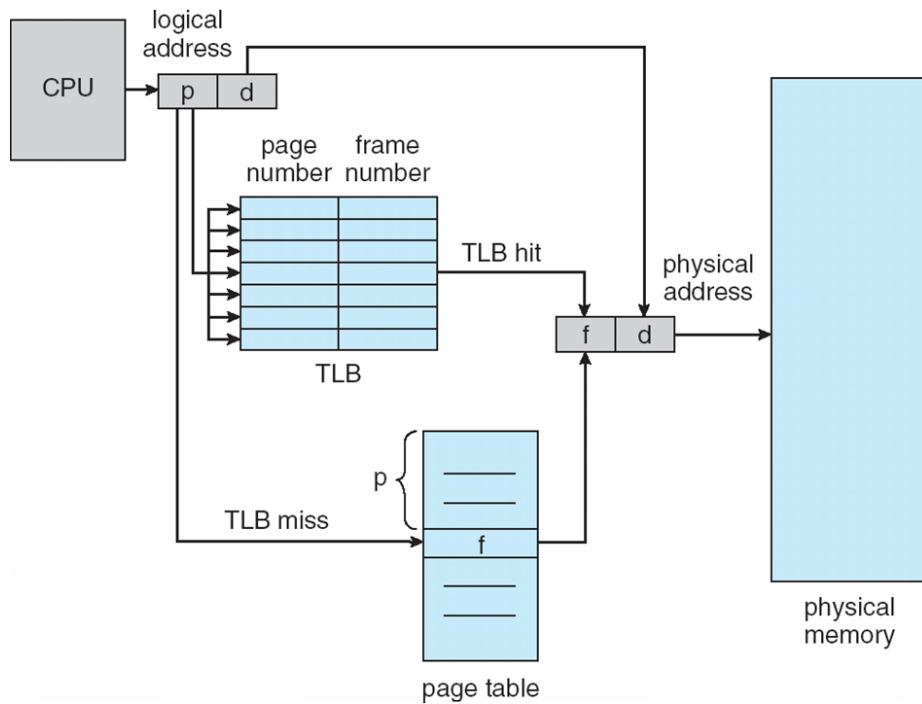
- 페이지 테이블은 메인 메모리에 저장
 - 페이지 테이블 기준 레지스터(PTBR, Page-Table Base Register)가 페이지 테이블을 가리킴
 - 페이지 테이블 길이 레지스터(PTLR, Page-Table Length Register)가 페이지 테이블의 크기를 표시
 - 매 데이터/명령 접근을 위해 두 번의 메모리 액세스를 요구
 - 한 번은 페이지 테이블, 또 한번은 데이터/명령
- 두 번의 메모리 접근 문제는 TLB(Translation Look-aside Buffer)라고 불리는 특수한 소형 하드웨어 캐시가 사용하여 해결
 - TLB는 매우 빠른 연관 메모리(associative memory)로 구성
 - TLB 내의 각 항목은 키(key)와 값(value)의 두 부분으로 구성

연관 메모리 (Associative Memory)

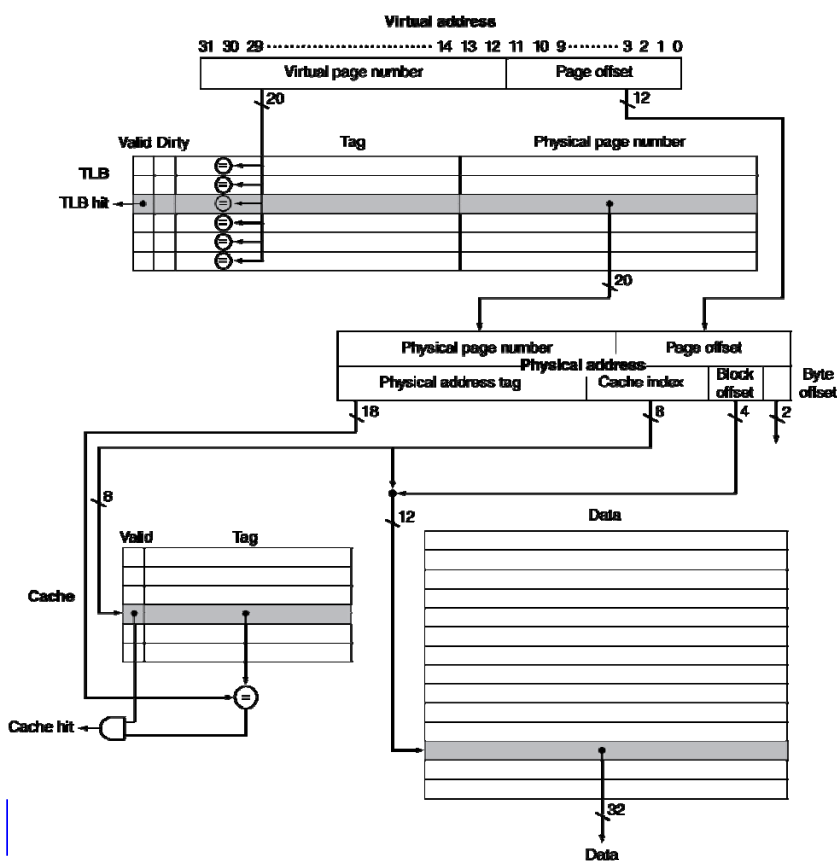
- TLB에서는 페이지 번호가 키가 되고, 프레임 번호가 값이 됨
 - 키가 주어지면 병렬로 검색
- 주소 변환 (A' , A'')
 - A' 가 TLB에 있으면 프레임 번호 가져옴
 - 없으면 페이지 테이블에서 프레임 번호 가져옴



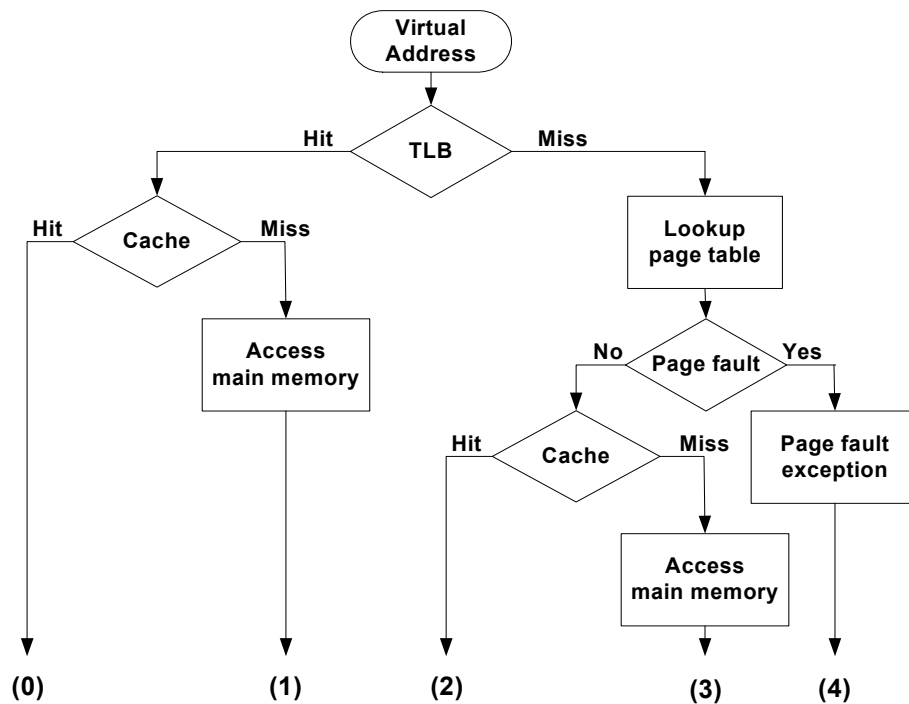
TLB를 이용한 페이징 하드웨어



보충 학습: 가상 메모리, TLB, 캐시의 통합 (1)



보충 학습: 가상 메모리, TLB, 캐시의 통합 (2)



실제 메모리 접근 시간 (Effective Memory Access time)

- ❑ 페이지 번호가 TLB에서 발견되는 비율은 **적중률(hit ratio)**
- ❑ 실제 메모리 접근 시간 예 (캐시 메모리 고려하지 않은 경우)
 - TLB 탐색 20 ns, 메모리 접근 100 ns, TLB 적중률 80%
 - TLB 적중 시 데이터를 위한 메모리 접근
 - TLB 실패 시 페이지 테이블 메모리 접근, 데이터 메모리 접근
 - 실제 메모리 접근 시간

$$= \text{적중률} \times \text{메모리 접근} + (1 - \text{적중률}) \times (2 \times \text{메모리 접근})$$

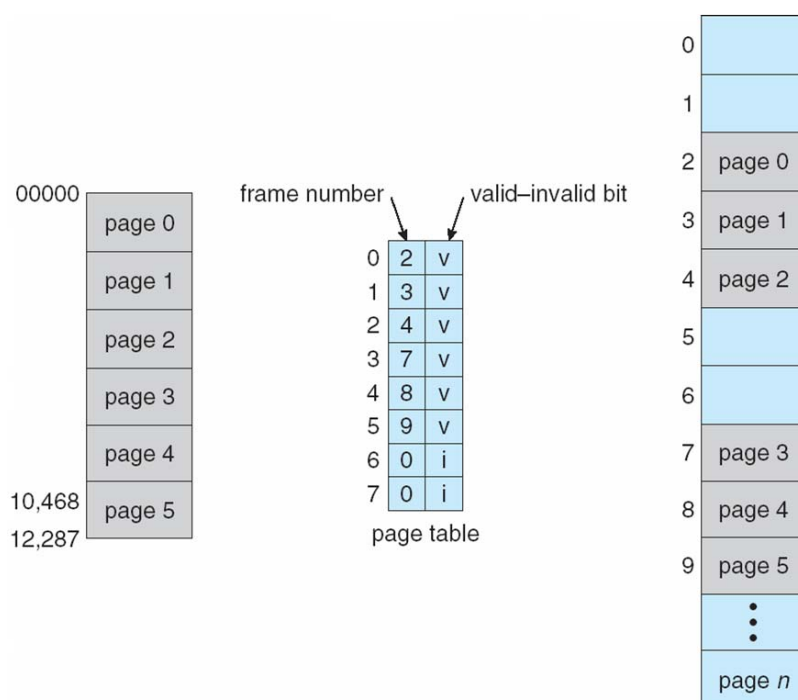
$$= 0.8 \times 100 + 0.2 \times (2 \times 100)$$

$$= 120 \text{ ns}$$

메모리 보호 (Memory Protection)

- ❑ 메모리 보호는 각 페이지에 붙어 있는 **보호 비트(Protection bit)**에 의해 구현
- ❑ 페이지 테이블의 각 항목에는 **유효/무효(valid/invalid)** 비트 추가
 - 이 비트가 **유효(valid)**로 설정되면 관련된 페이지가 프로세스의 합법적인 페이지임을 표시
 - 이 비트가 **무효(Invalid)**로 설정되면 그 페이지는 프로세스의 논리 주소 공간에 속하지 않는다는 것을 표시

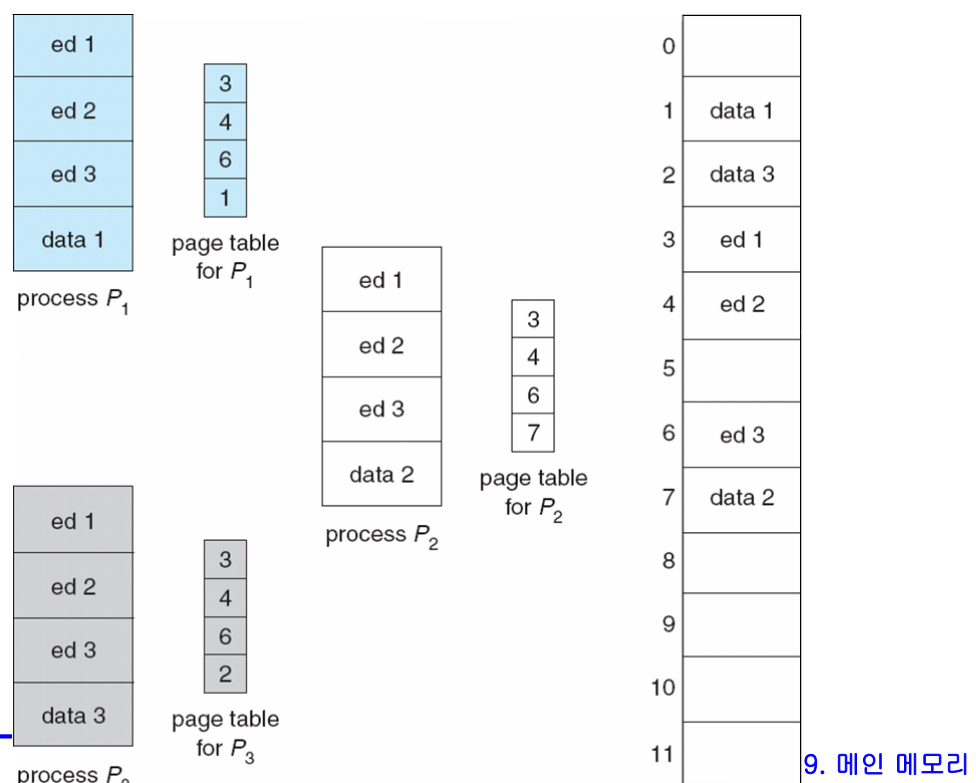
페이지 테이블에서의 유효/무효 비트



공유 페이지(Shared Page)

- ❑ 페이징의 또 다른 장점은 코드를 쉽게 공유할 수 있다는 점
- ❑ 재진입 가능 코드(reentrant code 또는 pure code)라면 공유가 가능
 - 재진입 가능 코드는 수행하는 동안 절대로 (프로그램 내용이) 변하지 않는 코드
 - 따라서 두 개나 그 이상의 프로세스들이 동시에 같은 코드를 수행할 수 있음
- ❑ 공유되는 프로그램
 - 문서 편집기, 컴파일러, 윈도우 시스템, 실시간 라이브러리, 데이터베이스 시스템 등

페이징 환경에서 코드 공유

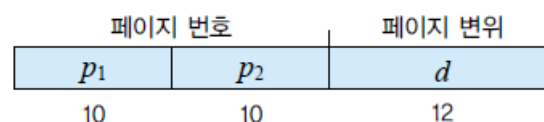


페이지 테이블의 구조 (Structure of the Page Table)

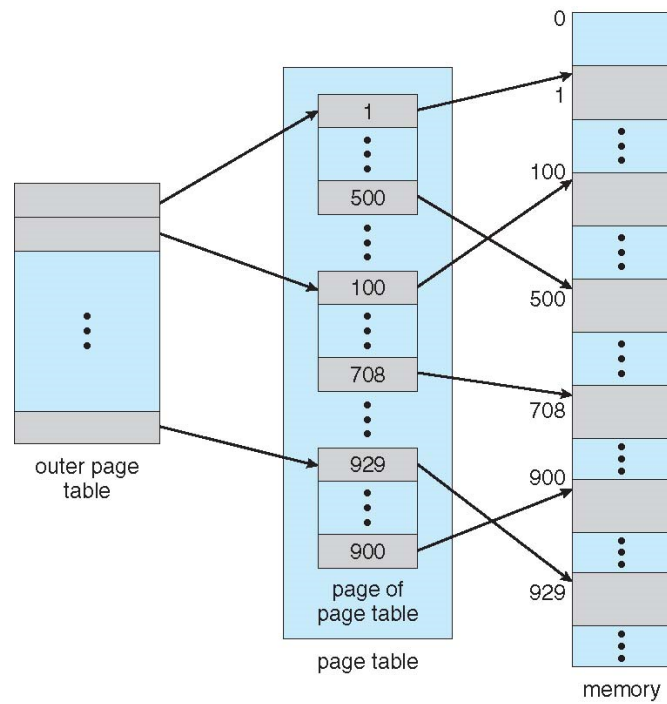
- 페이징 직접 적용 시 **페이지 테이블 크기가 커짐**
 - 32비트 논리주소 공간, 4KB(2^{12}) 페이지 크기 예
 - 페이지 테이블을 1백만개 엔트리 ($2^{32}/2^{12}$)
 - 각 페이지 테이블 엔트리 당 4바이트라면 페이지 테이블 크기는 4MB
- 페이지 테이블을 구성하는 가장 일반적인 방법을 소개
 - 계층적 페이징 (Hierarchical Paging)
 - 해시형 페이지 테이블(Hashed Page Table)
 - 역 페이지 테이블(Inverted Page Table)

계층적 페이징 (Hierarchical Paging)

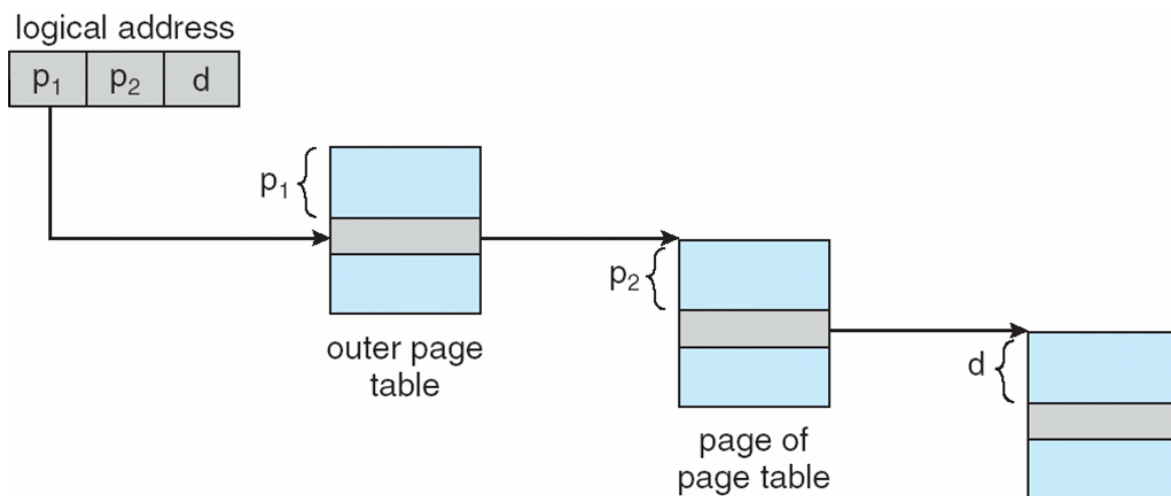
- 논리 주소 공간을 여러 단계의 페이지 테이블로 분할
 - 두 단계 페이징 기법(two-level paging scheme)이 한 예
- 두 단계 페이징 기법(two-level paging scheme) 예
 - 32비트 논리주소 공간, 4 KB(2^{12}) 페이지 크기의 시스템 논리주소
 - 20 비트 페이지 번호(2^{20} 항목), 4MB ($2^{20} \times 4$ 바이트/항목) 크기
 - 12 비트 페이지 변위
 - 페이지 테이블도 페이지화되면 페이지 번호가 분할
 - 10비트 페이지 번호 (2^{10} 항목)
 - 10비트 페이지 변위 (2^{10} 항목)



두 단계 페이지 테이블 기법



두 단계 페이지 주소 변환



64 비트 논리 주소 공간

- ❑ 64 비트 논리 주소 공간을 가진 시스템에서는 2 단계 페이징 기법이 적절하지 못함
- ❑ 페이지 크기 4 KB (2^{12}) 라 가정
 - 페이지 테이블은 2^{52} 항목으로 구성
 - 주소 구성

| outer page | inner page | page offset |
|------------|------------|-------------|
| p_1 | p_2 | d |
| 42 | 10 | 12 |

- 바깥 페이지 테이블은 2^{42} 항목으로 구성
 - 페이지 테이블 크기가 너무 큼
- 두 번째 바깥 페이지를 도입한 3 단계 페이지 테이블 구성
 - 두 번째 바깥 페이지도 여전히 큼, 2^{32}
 - 물리 메모리 접근을 위해 4번의 메모리 접근도 문제

3-단계 페이지 주소

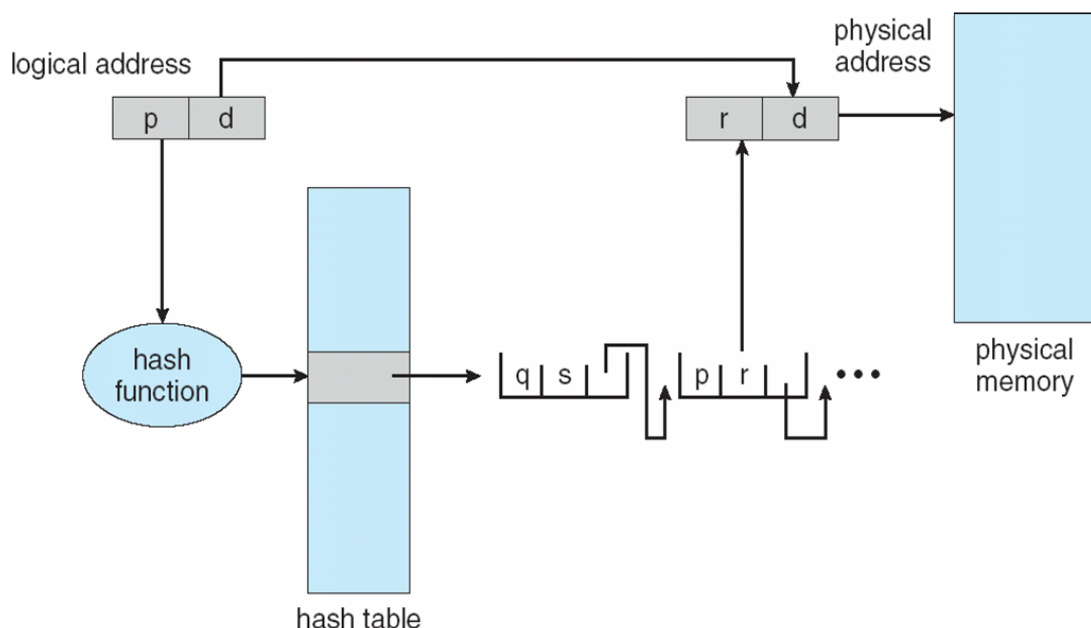
| outer page | inner page | offset |
|------------|------------|--------|
| p_1 | p_2 | d |
| 42 | 10 | 12 |

| 2nd outer page | outer page | inner page | offset |
|----------------|------------|------------|--------|
| p_1 | p_2 | p_3 | d |
| 32 | 10 | 10 | 12 |

해시형 페이지 테이블 (Hashed Page Table)

- 주소 공간이 32 비트보다 커지면 **해시형 페이지 테이블**을 사용
 - 가상 페이지 번호가 해시 값이 되어 페이지 테이블 참조
 - 같은 위치로 해시되는 **충돌(collision)** 인 경우를 위해 각 항목은 연결 리스트로 구성
- 해시형 페이지 테이블 동작
 - 가상 주소 공간으로부터 페이지 번호로 해싱
 - 해시형 페이지 테이블에서 연결 리스트를 따라가며 페이지 번호 비교
 - 일치하면 그에 대응하는 페이지 프레임 번호를 획득
- 64 비트 시스템에서는 변형된 **클러스터 페이지 테이블** 사용
 - 해시 페이지 테이블의 각 항목이 한 개의 페이지가 아닌 여러 페이지 (예를들어 16)를 가리킴
 - 성긴(sparse) 주소 공간에 유용 (메모리 접근이 비연속적이고 전 주소 공간에 넓게 퍼져 있는 경우)

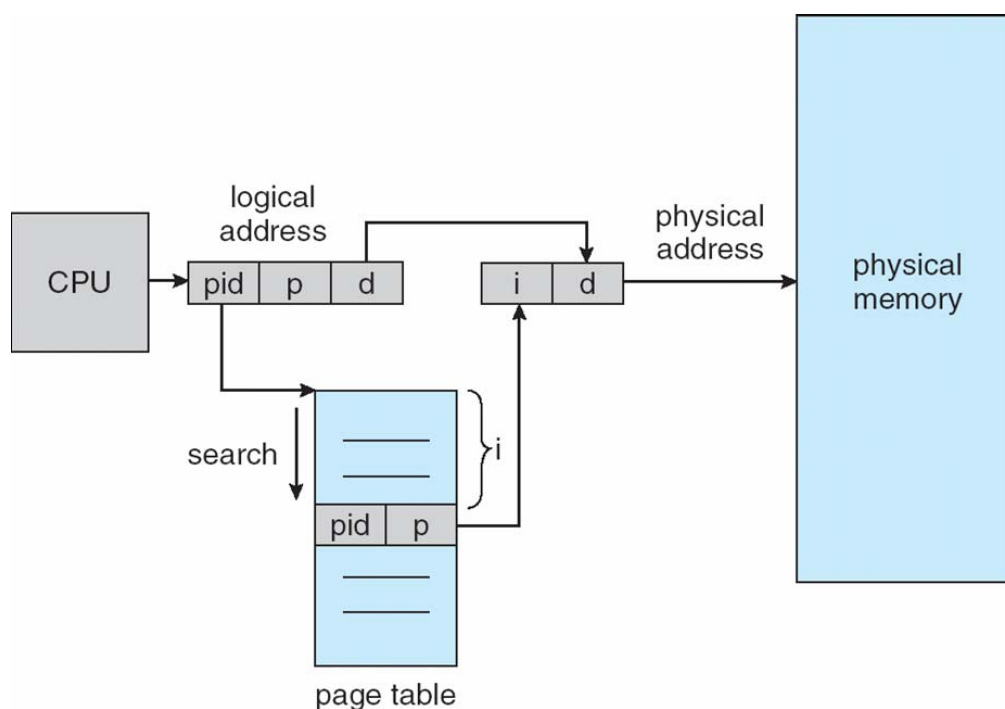
해시된 페이지 테이블



역 페이지 테이블 (Inverted Page Table)

- 지금까지 페이지 테이블은 페이지 마다 하나의 항목을 가짐
 - 페이지 테이블이 가상 주소에 대해 오름차순으로 정렬
 - 페이지 테이블의 크기가 커짐
- 역 페이지 테이블(inverted page table)
 - 메모리 프레임마다 한 항목씩을 할당
 - 논리 페이지마다 항목을 가지는 대신 물리 프레임에 대응되는 항목만 테이블에 저장하기 때문에 메모리에서 훨씬 작은 공간을 점유
 - 프레임에 따라 저장되어 있어 이 테이블에 대한 탐색 시간 필요
 - 주소 변환시간이 길어짐
 - 이 시간을 줄이기 위하여, 페이지 테이블을 해시(hash)
 - 또한 최근에 사용된 항목들을 버리지 말고 TLB 연관 메모리에 저장

역 페이지 테이블 구조



예: Intel 32비트와 64비트 구조

- ❑ 현재 가장 널리 사용되는 CPU 칩
- ❑ Pentium CPU는 32비트로 IA-32 구조(architecture)라고 함
- ❑ 현재 Intel CPU는 64비트로 IA-64 구조하고 함
- ❑ 여기서는 주요 메모리 관리 개념만 소개

예: IA-32 구조 (1)

- ❑ 세그먼테이션과 페이지화된 세그먼테이션(segmentation with paging)을 모두 지원
 - 각 세그먼트는 최대 4 GB
 - 한 프로세스 당 16 K 개 세그먼트를 가질 수 있음
 - 각 프로세스의 주소 공간은 두 개의 분할
 - 첫 번째 분할은 그 프로세스가 독점적으로 사용하는 8 K개 세그먼트
 - 지역 기술자 테이블(LDT, Local Descriptor Table)에 저장
 - 두 번째 분할은 모든 프로세스 사이에서 공유가 가능한 8 K개 세그먼트
 - 전역 기술자 테이블(GDT, Global Descriptor Table)에 저장

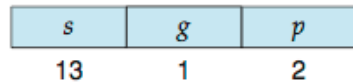
예: IA-32 구조 (2)

□ 논리 주소는 셀렉터와 변위(selector, offset)의 쌍으로 구성

- 셀렉터 16비트
- 변위 32비트

□ 셀렉터

- 다음과 같은 16 비트 수로 구성

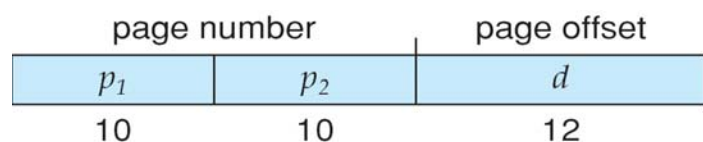
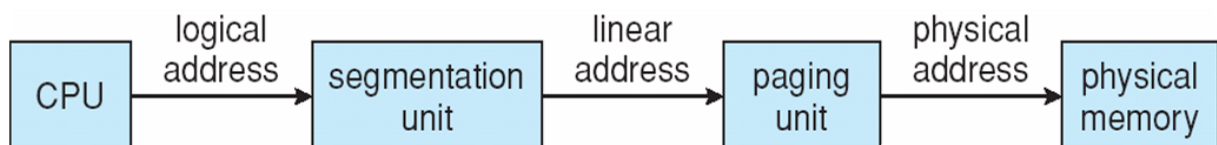


- *s*는 세그먼트 번호
- *g*는 세그먼트가 GDT인지 LDT인지를 표시,
- *p*는 보호(Protection)와 관련된 정보를 표시
- 6개의 16비트 세그먼트 레지스터(CS, DS, SS, ES, FS, GS)에 저장
 - 한 프로세스는 최대 6개의 세그먼트를 가리킴

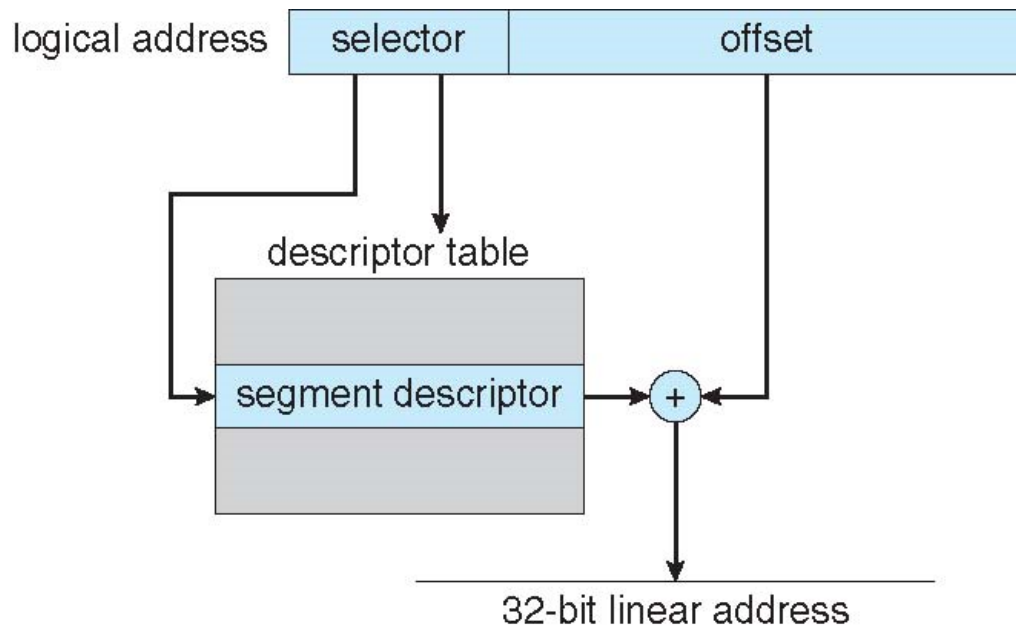
예: IA-32 구조 (3)

□ IA-32 구조 주소 변환

- CPU는 논리 주소 생성
- 세그멘테이션 유닛은 각각의 논리 주소를 32 비트 선형 주소(linear address)로 변환
- 페이징 유닛은 선형 주소를 메인 메모리의 물리 주소로 변환
 - 페이지 크기는 4 KB 또는 4 MB



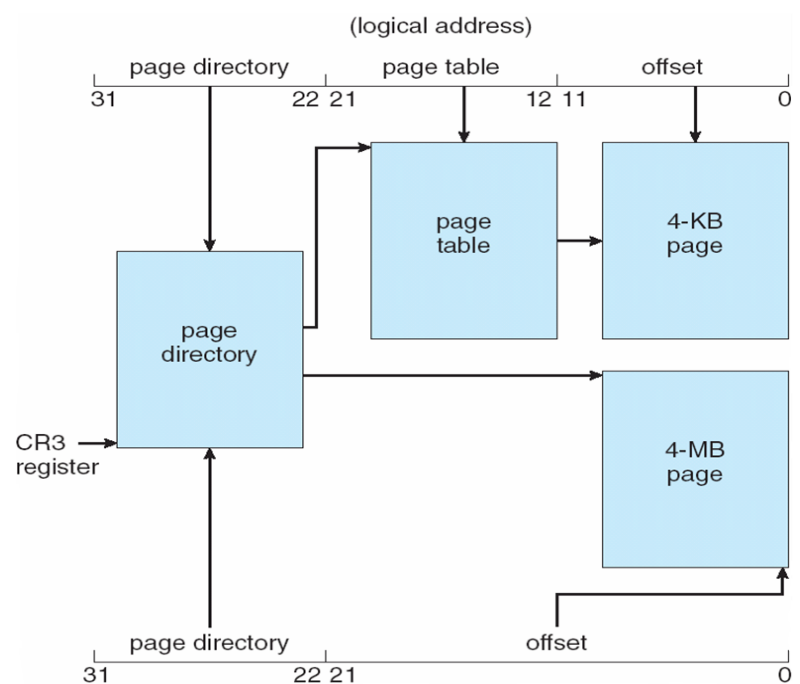
Intel IA-32 세그멘테이션



Intel IA-32 페이징

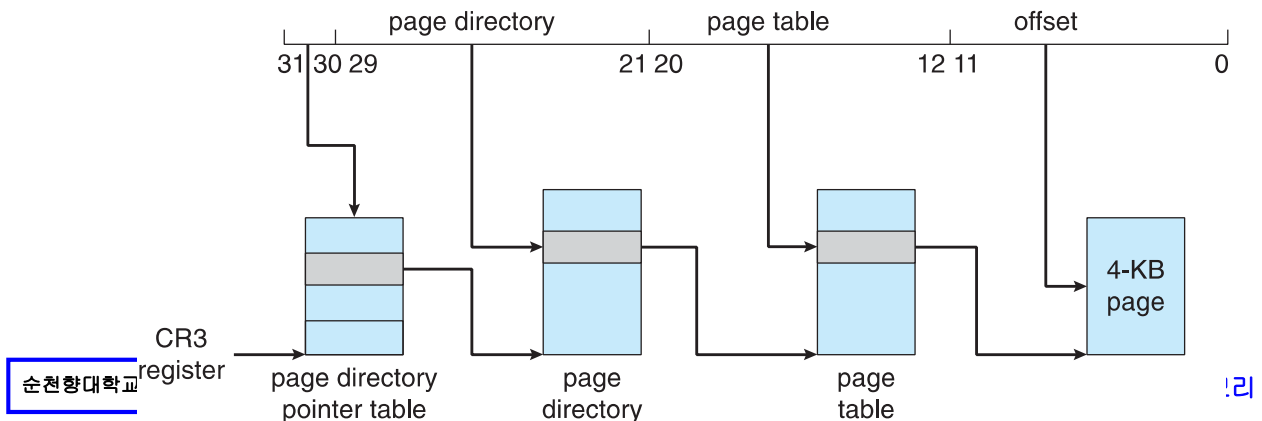
□ 2 단계 페이징 기법

- 4 KB 크기의 페이지를 사용할 때 적용
- p_1 은 페이지 디렉토리 (page directory)라고 부르는 최상위 페이지 테이블의 항목을 가리킴
 - CR3 레지스터는 현재 프로세스의 페이지 디렉토리를 가리킴
- p_2 는 하위 페이지 테이블의 변위 표시



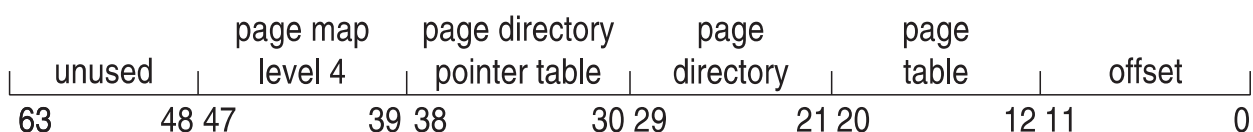
IA-32 페이지 주소 확장 (Page Address Extension, PAE)

- 32비트 물리 주소의 4 GB 메모리 공간 제한이 문제가 되어 32비트 주소로 4 GB 보다 큰 물리 주소 공간 접근 가능
 - 3 단계 페이징 기법
 - 최상위 2비트는 **페이지 디렉토리 포인터 테이블**
 - 페이지 디렉토리 와 페이지 테이블의 항목 길이는 32비트에서 64비트로 변경
 - 전체 36비트 주소 공간으로 확장하여 64 GB 물리 메모리 접근
 - 프레임 기준 주소 24 비트
 - 12비트 오프셋



Intel x86-64

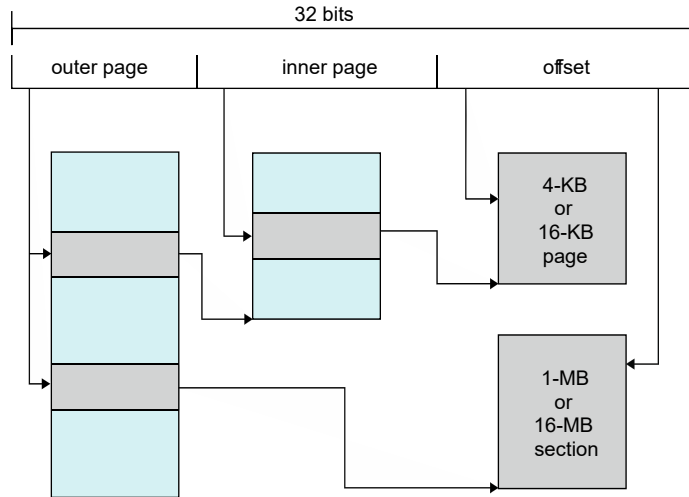
- 현재 사용 중인 **인텔 x86 아키텍처**
 - AMD 개발하여 Intel이 채택
- 64비트는 너무 커서(> 16 exabytes) 48비트만 사용
 - 페이지 크기는 4 KB, 2 MB, 1 GB
 - 4 단계의 페이징 계층
 - PAE를 사용하여 48 비트 가상 주소로 52 비트 물리 주소 공간 지원할 수 있음



ARM 아키텍처 예

□ 대부분의 모바일 플랫폼 CPU로 사용

- 저전력 32 비트 CPU
- 4 KB, 16 KB 페이지
- 1 MB, 16 MB 페이지 (섹션이라고 함)
- 섹션에 대해서는 1-단계, 작은 페이지에 대해서는 2-단계
- 두 단계 TLB
 - 외부 레벨은 2개의 마이크로 TLB (데이터, 명령)
 - 내부 레벨은 단일 주 TLB
 - 먼저 내부 레벨이 조사되고, 미스 시 외부 레벨이 조사
 - 외부도 미스 시 페이지 테이블 조사



연습문제

- 연습문제 9.1
- 연습문제 9.4
- 연습문제 9.6
- 연습문제 9.9