8장. 주 메모리 (Main Memory)

순천향대학교 컴퓨터공학과
이 상 정

강의 목표 및 내용

- 목표
  - 메모리 하드웨어를 구성하는 다양한 방법을 기술
  - 페이징과 세그먼테이션 기법을 포함한 다양한 메모리 관리 기법들을 설명
- 내용
  - 배경
  - 스와핑
  - 연속 메모리 할당
  - 페이징
  - 세그멘테이션
  - Intel Pentium과 Linux 사례

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)

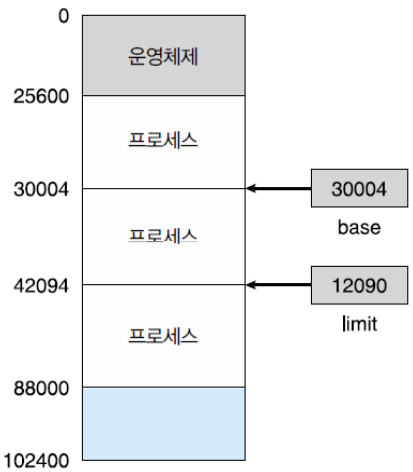


그림 8.1 베이스와 상한 레지스터가 논리 주소 공간을 정의한다.

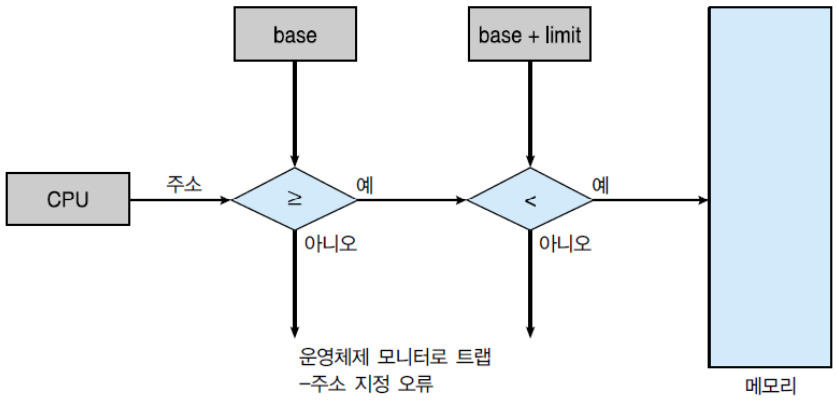


그림 8.2 베이스와 상한 레지스터를 통한 하드웨어 주소 보호

주소의 할당 (Address Binding)

- 프로그램이 실행되기 위해서는 주 메모리로 올라와서 프로세스가 되어야 함
  - 프로그램은 원래 이진 실행 파일 형태로 디스크에 저장
- 입력 큐(input queue)
  - 디스크에서 주 메모리로 들어오기를 기다리고 있는 프로세스들의 집합
  - 큐에서 하나의 프로세스를 선택해서, 메모리로 올린 후 실행하고 이 프로세스는 실행을 하는 동안 메모리에서 명령어와 자료를 액세스
- 사용자 프로그램은 실행되기 전에 여러 단계를 거침

메모리 주소 공간에서 명령어와 자료의 바인딩

- 메모리 주소 공간에서 명령어와 자료의 바인딩은 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 코드를 생성할 수 있음
    - 만일 이 위치가 변경되어야 한다면 이 코드는 다시 컴파일되어야 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 중간에 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

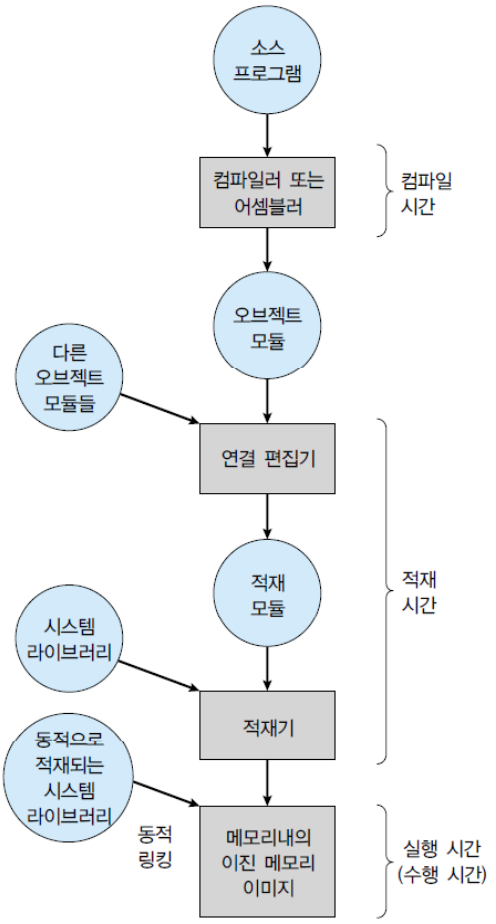


그림 8.3 사용자 프로그램의 단계별 처리 과정

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 관리기 (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
- 사용자 프로그램은 논리 주소만을 다루고 실제적인 물리 주소를 결코 알 수 없음

재배치 레지스터를 이용한 동적 재배치

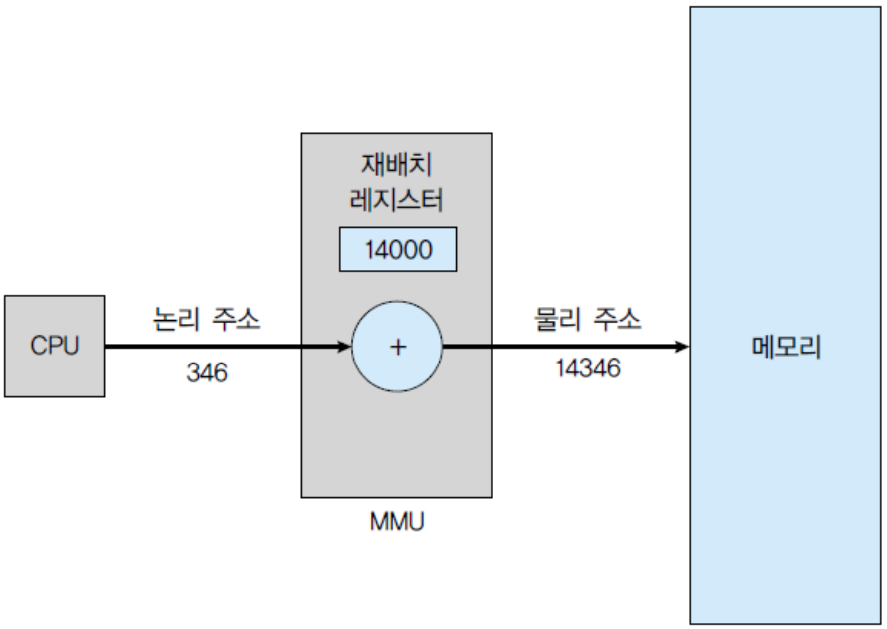


그림 8.4 재배치 레지스터를 이용한 동적 재배치

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 적재도지 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 루틴을 제공 가능

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

- 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 스텁(stub)을 사용하여 메모리에 적재된 라이브러리의 위치를 찾음
  - 메모리에 없으면 디스크에서 가져옴
  - 스텁 자신을 찾은 루틴의 번지로 대체하고, 실행

스와핑 (Swapping)

- 스와핑
  - 프로세스는 실행도중에 임시로 보조 메모리(디스크)로 보내어졌다가 다시 메모리로 되돌아 올 수 있음
- 스와핑은 우선순위 기반 스케쥴링 알고리즘에 적용될 수 있음
  - 낮은 우선순위 프로세스를 디스크로 스왑
  - 높은 우선순위 프로세스가 끝나면, 낮은 우선순위 프로세스는 다시 메모리로 스왑
  - 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

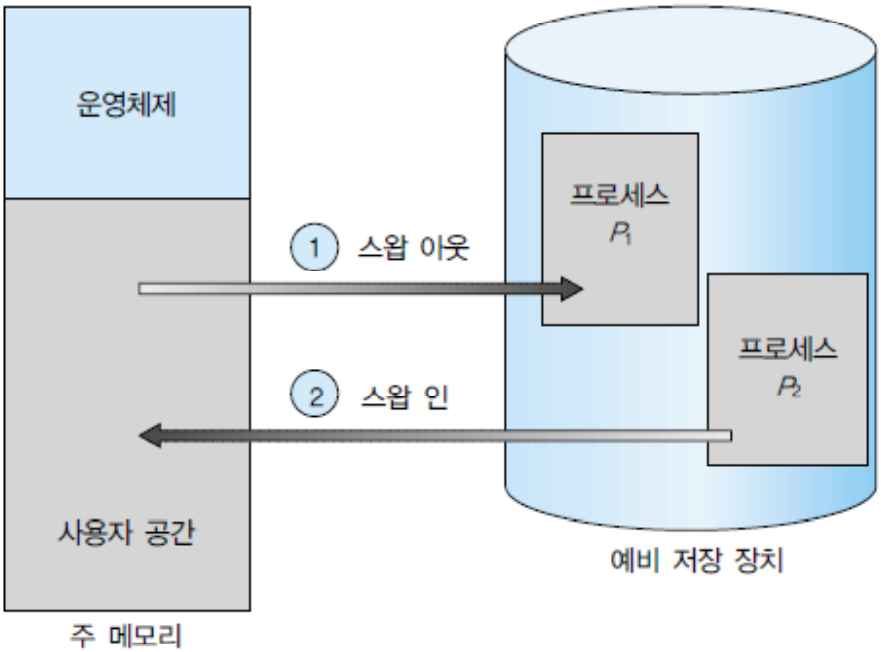


그림 8.5 디스크를 사용한 두 프로세스의 스와핑

연속 메모리 할당 (1) (Contiguous Memory Allocation)

- 메모리는 일반적으로 두 개의 부분으로 분할
  - 메모리에 상주하는 운영체제로, 일반적으로 인터럽트 벡터와 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 레지스터에 의해 수행

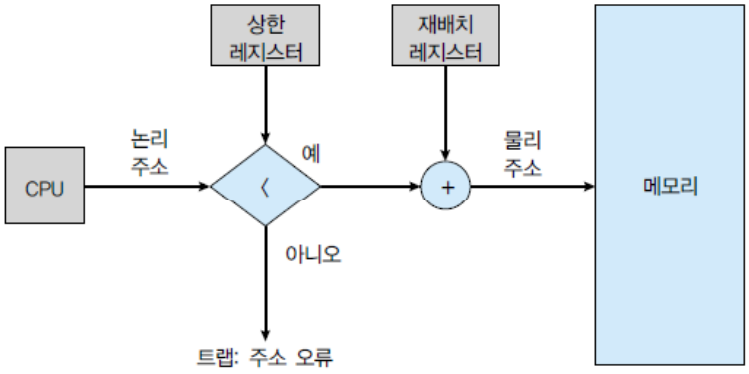
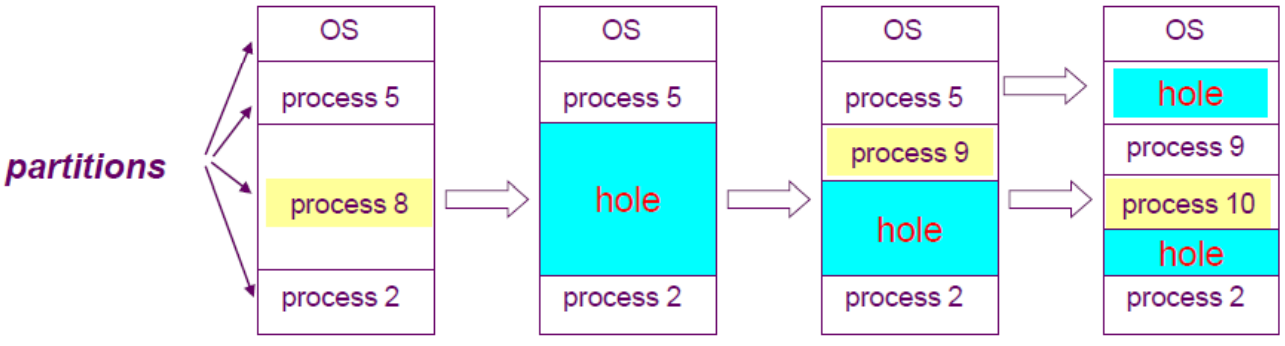


그림 8.6 재배치와 상한 레지스터를 지원하는 하드웨어

연속 메모리 할당 (2)

- 메모리 할당 (memory allocation)
  - 고정된 크기로 분할(single partition)
  - 서로 다른 크기로 분할(변수 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)
    - 첫번째 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 좋다는 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

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 멱승으로 512바이트에서 16M 바이트 범위
  - 논리 메모리는 페이지(page)라 불리는 같은 크기의 블록으로 분할
  - 한 프로세스가 실행될 때 프로세스의 페이지는 보조 메모리로부터 주 메모리 프레임으로 들어 감
  - 모든 자유 프레임 리스트들은 추적 관리
  - 논리 주소에서 물리 주소로 변환하는 페이지 테이블(page table) 필요
  - 내부 단편화

페이징 예

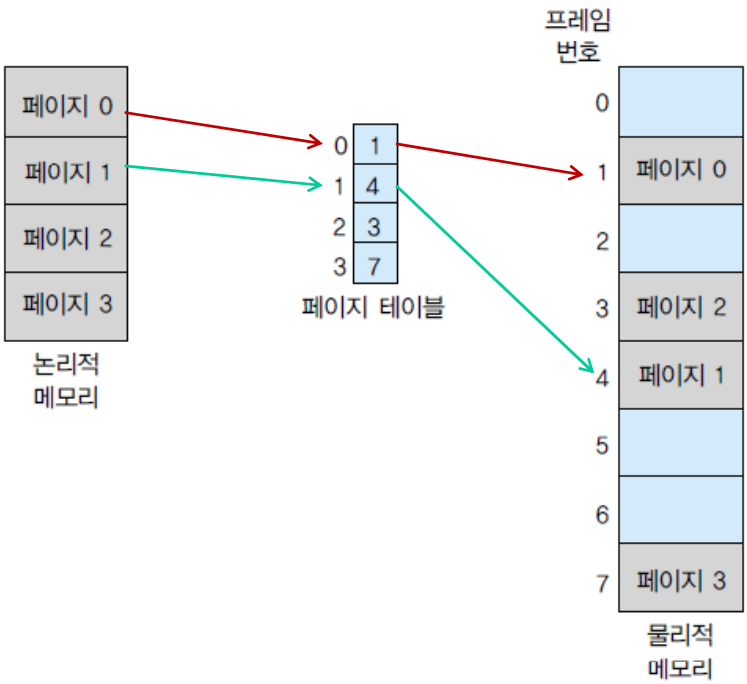
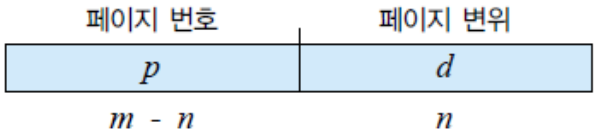


그림 8.8 논리 및 물리 메모리의 페이징 모델

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^m이고, 페이지가 2^n 크기라면 논리 주소의 상위 m − n 비트는 페이지 번호를 나타내고, 하위 n 비트는 페이지 변위를 나타냄



페이징 하드웨어

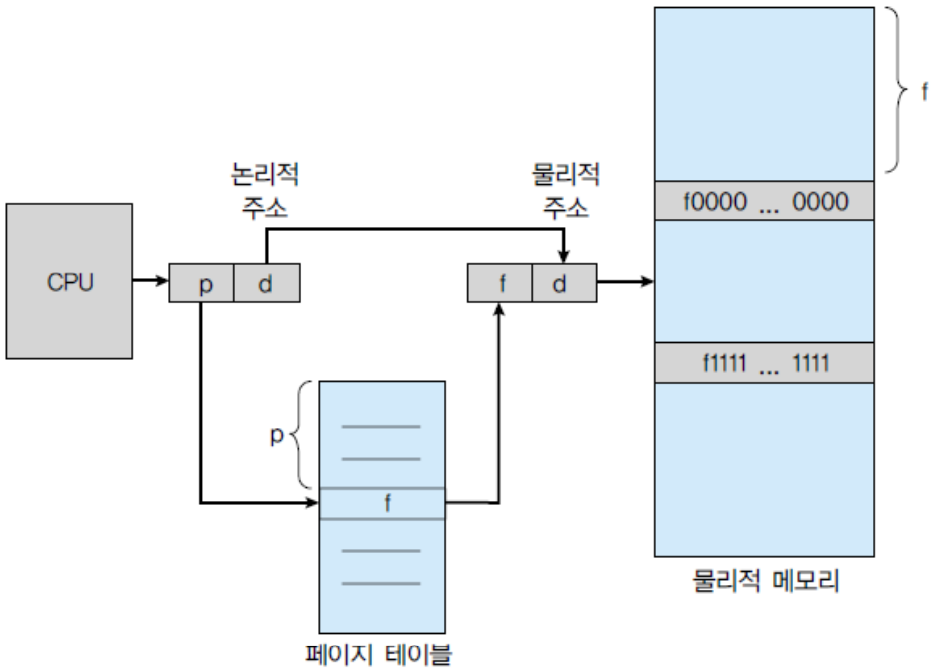
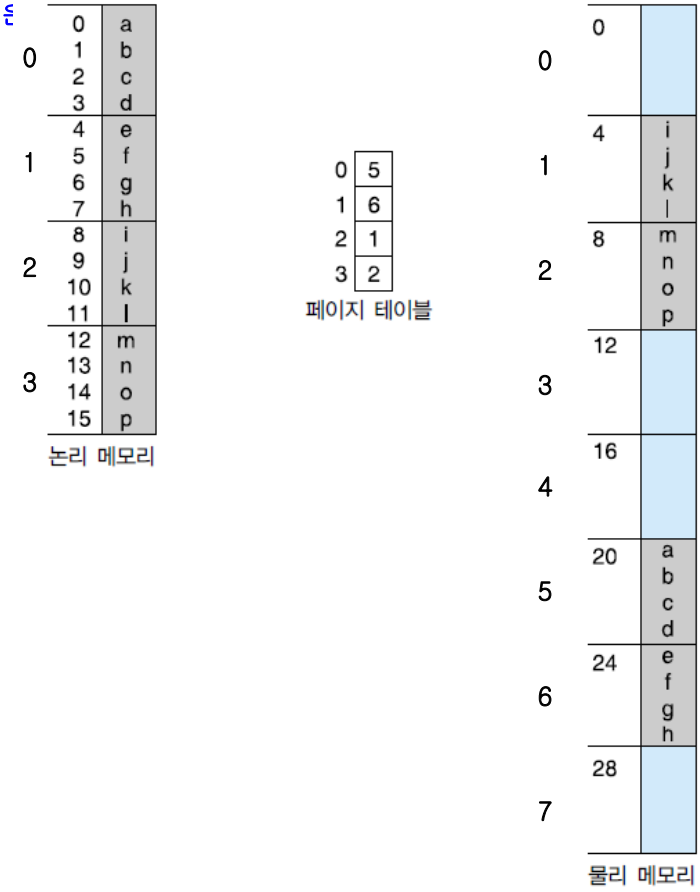


그림 8.7 페이징 하드웨어

주소변환 예



페이지 번호	페이지 변위
p	d
2	2

그림 8.9 4 바이트 페이지를 가진 32 바이트 메모리의 페이징 예

자유 프레임 (Free Frame)

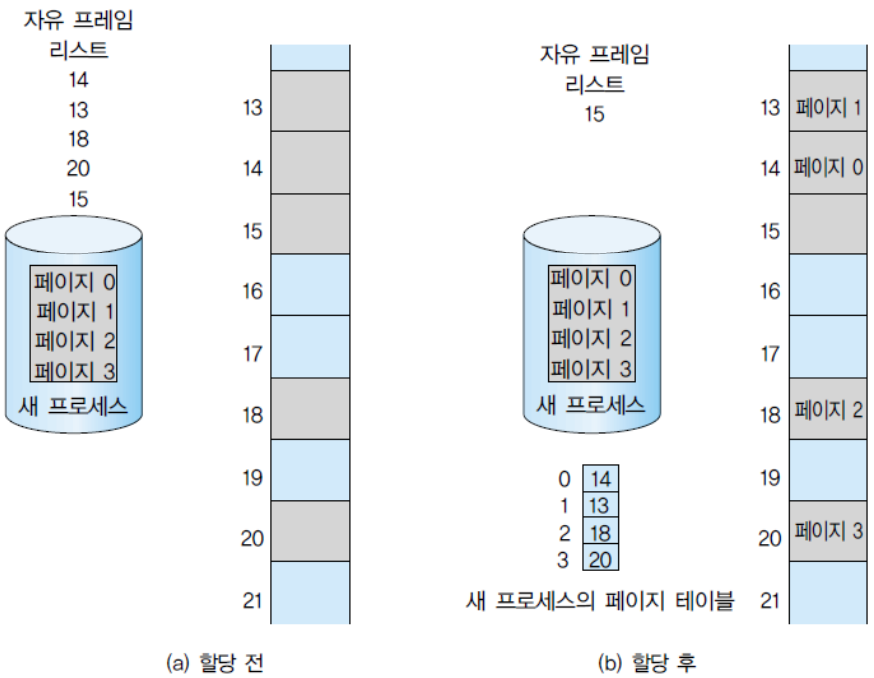


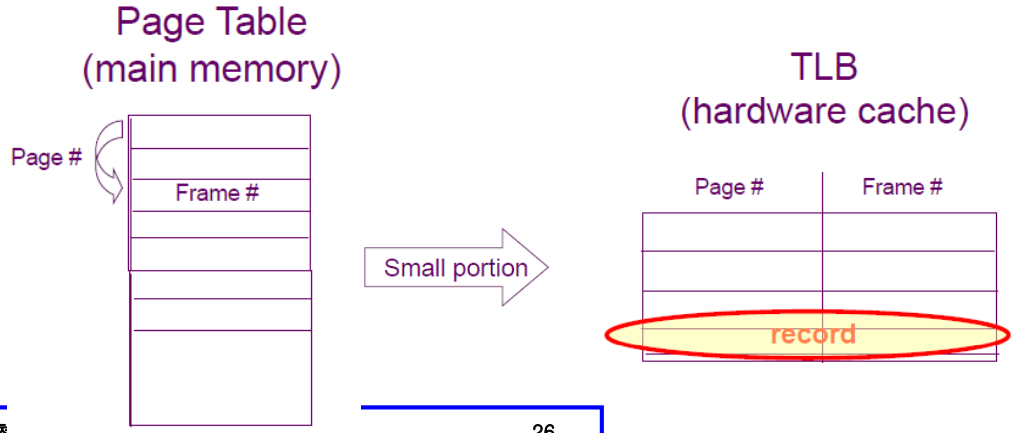
그림 8.10 자유 프레임

페이지 테이블 하드웨어

- 페이지 테이블은 주 메모리에 저장
  - 페이지 테이블 크기는 1M 항목 정도의 크기
  - 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

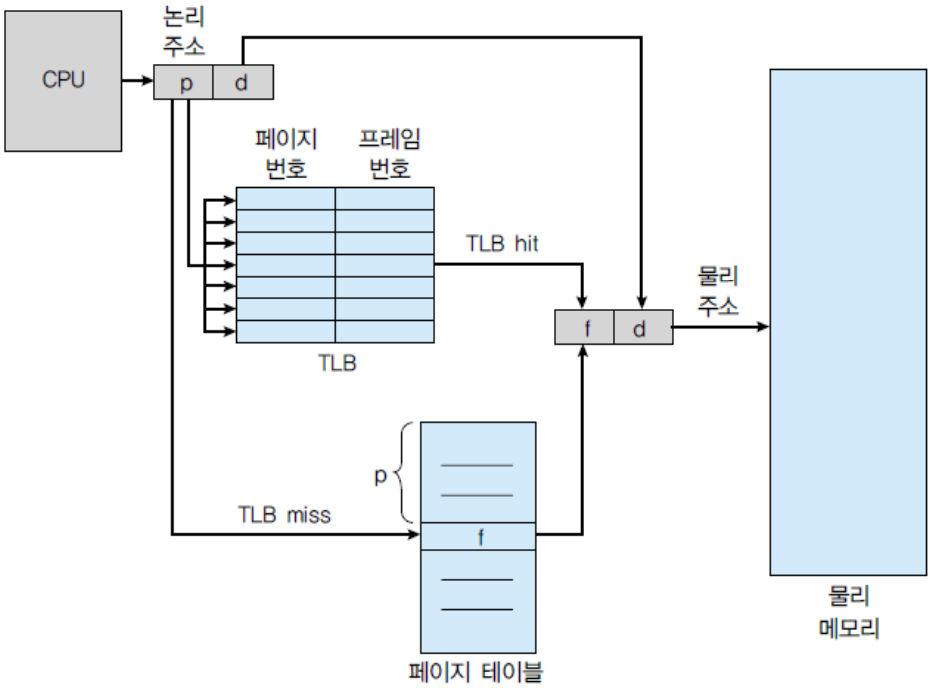


그림 8.11 TLB를 이용한 페이징 하드웨어

유효 메모리 접근 시간 (Effective Memory Access time)

- 페이지 번호가 TLB에서 발견되는 비율은 적중률(hit ratio)
- 유효 메모리 접근 시간 예
  - TLB 탐색 20 ns, 메모리 접근 100 ns, TLB 적중률 80%
  - 유효 메모리 접근 시간
    = 적중률 x (TLB 탐색 + 메모리 접근)
    + (1−적중률) x (TLB 탐색 + 2 x 메모리 접근)
    = 0.8 x (20 + 100) + 0.2 x (20 + 200)
    = 140 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

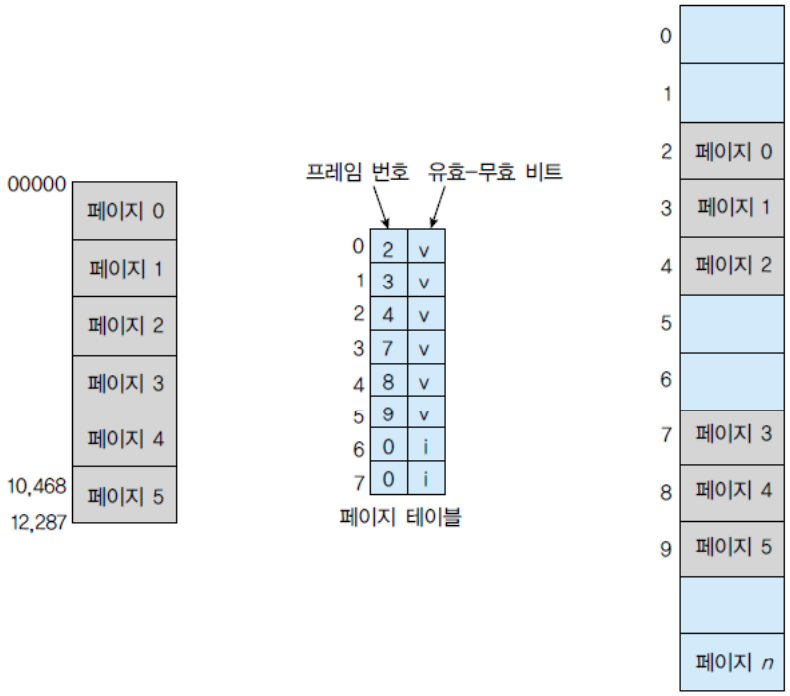


그림 8.12 페이지 테이블에서 유효(v) / 무효(i) 비트

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 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

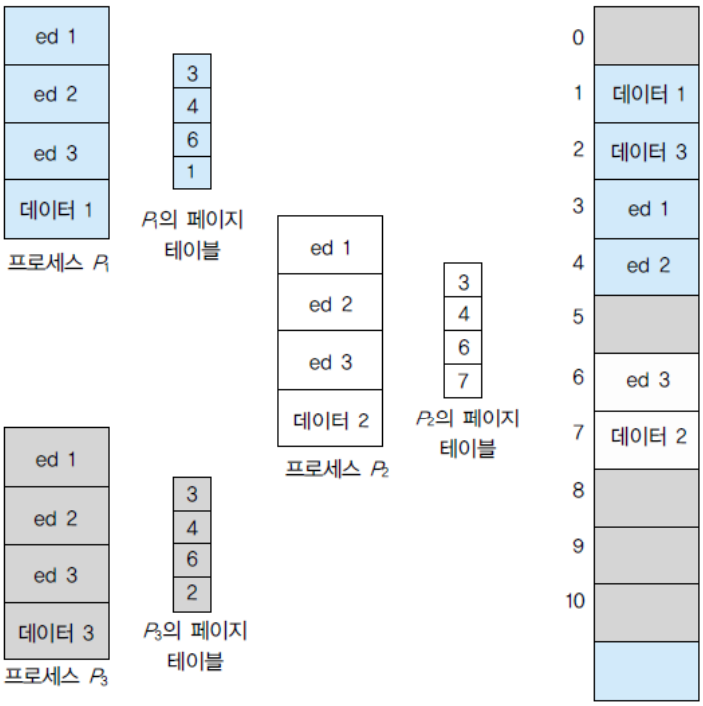


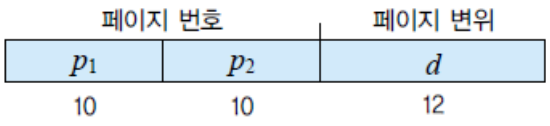
그림 8.13 페이징 환경에서의 코드 공유

페이지 테이블의 구조 (Structure of the Page Table)

- 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} X 4바이트/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

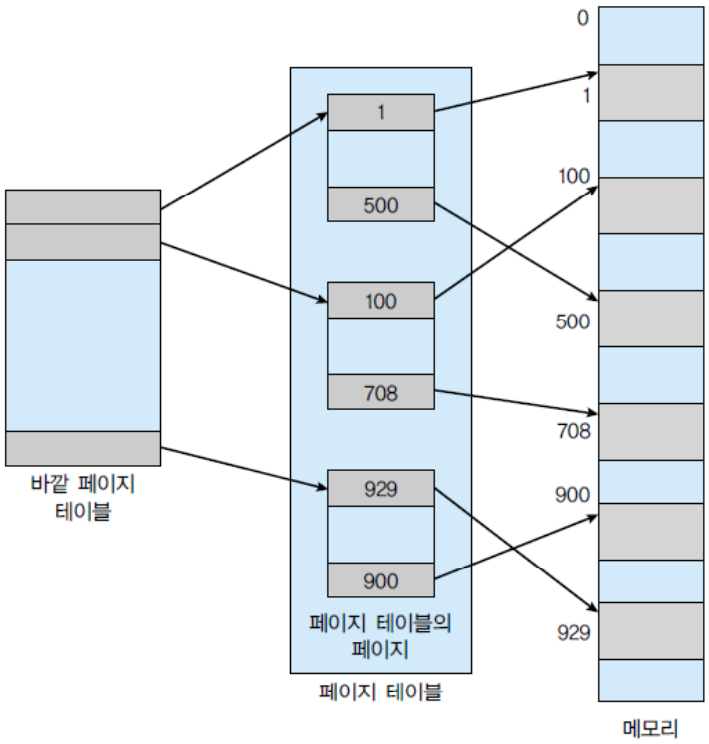


그림 8.14 두 단계 페이지 테이블 기법

두 단계 페이지 주소 변환

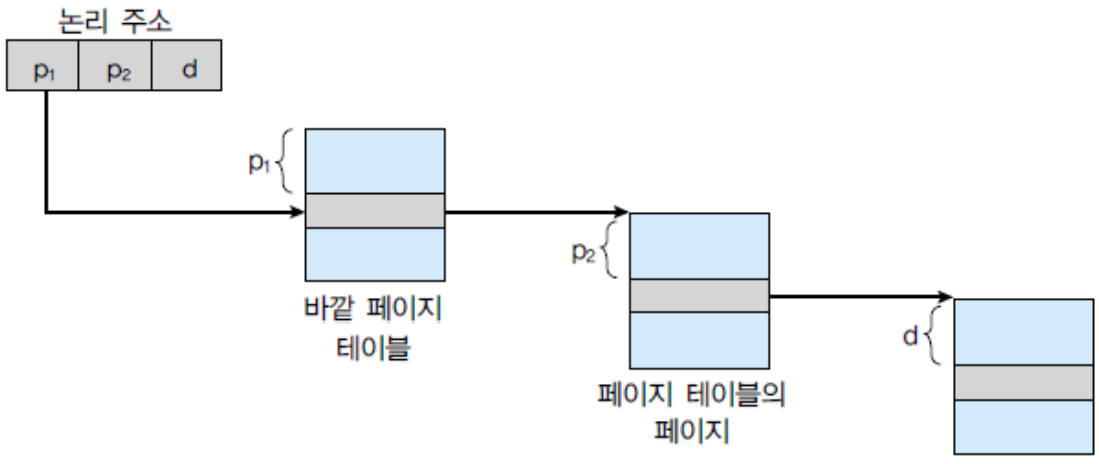


그림 8.15 2단계 32비트 페이징 구조에서의 주소 변환

해시형 페이지 테이블 (Hashed Page Table)

- 주소 공간이 32 비트보다 커지면 해시형 페이지 테이블을 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

해시된 페이지 테이블

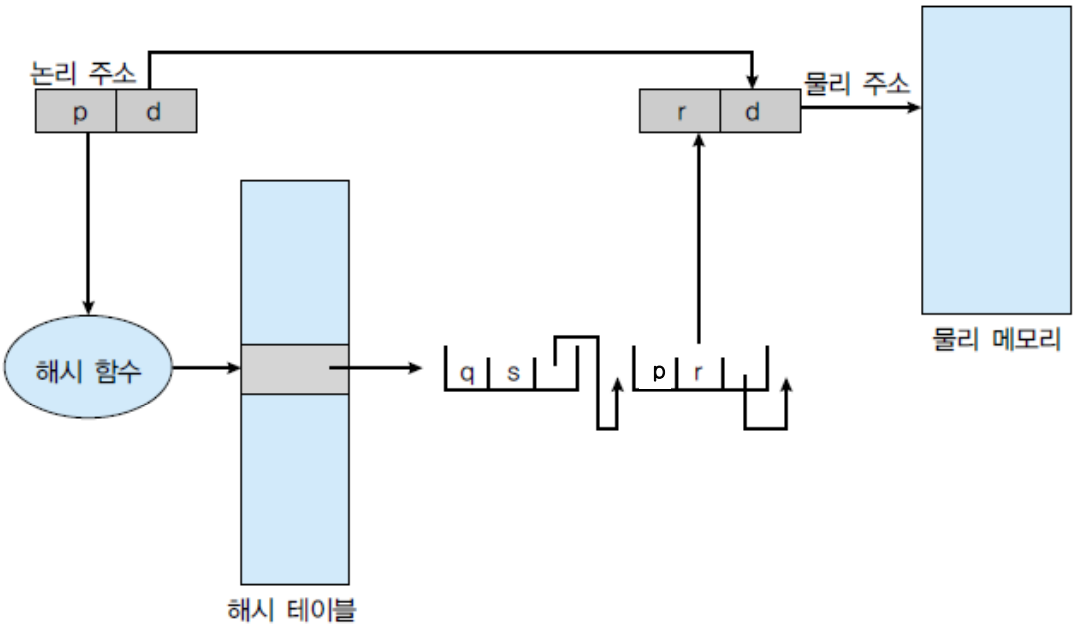


그림 8.16 해시된 페이지 테이블

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

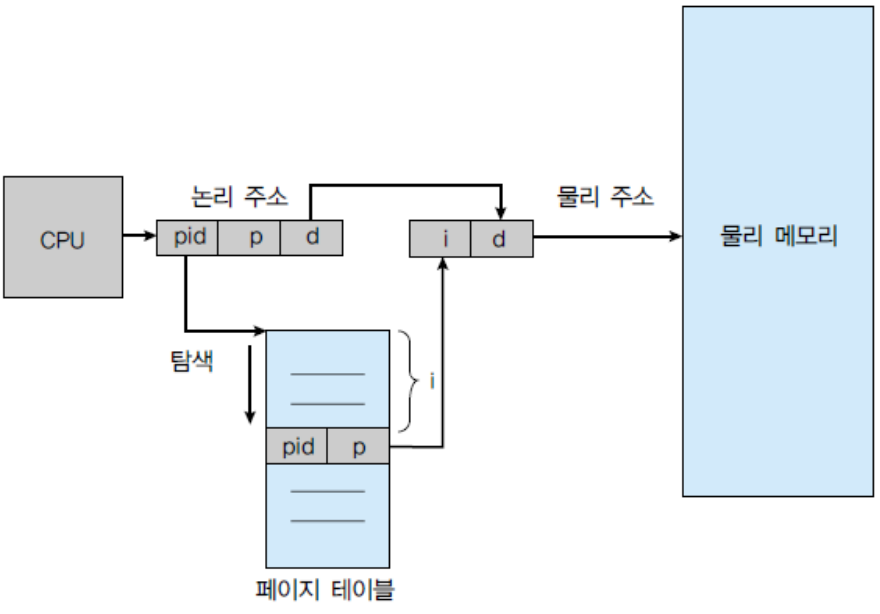


그림 8.17 역 페이지 테이블

세그먼테이션 (Segmentation)

- 사용자 관점에서 지원되는 메모리 관리 방식

- 프로그램은 세그먼트의 집합
- 세그먼트는 다음과 같은 것들의 논리적 단위
  - 주 프로그램 (main())
  - 프로시저
  - 함수
  - 메소드
  - 객체
  - 광역변수, 지역 변수
  - 스택
  - 심볼 테이블, 배열

프로그램의 사용자 관점

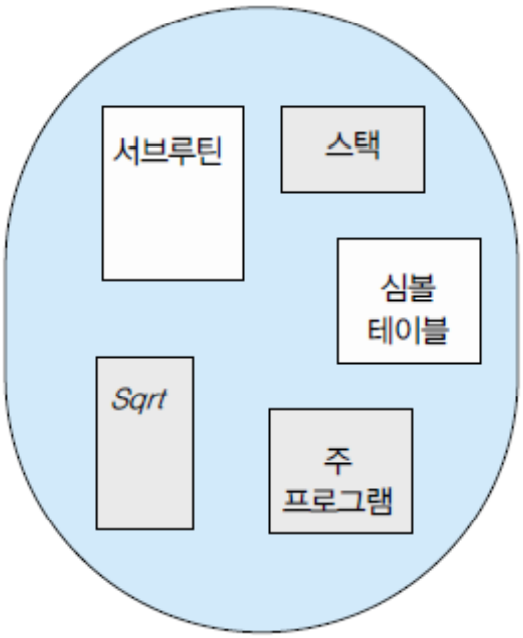


그림 8.18 프로그램의 사용자 관점

세그먼트 예

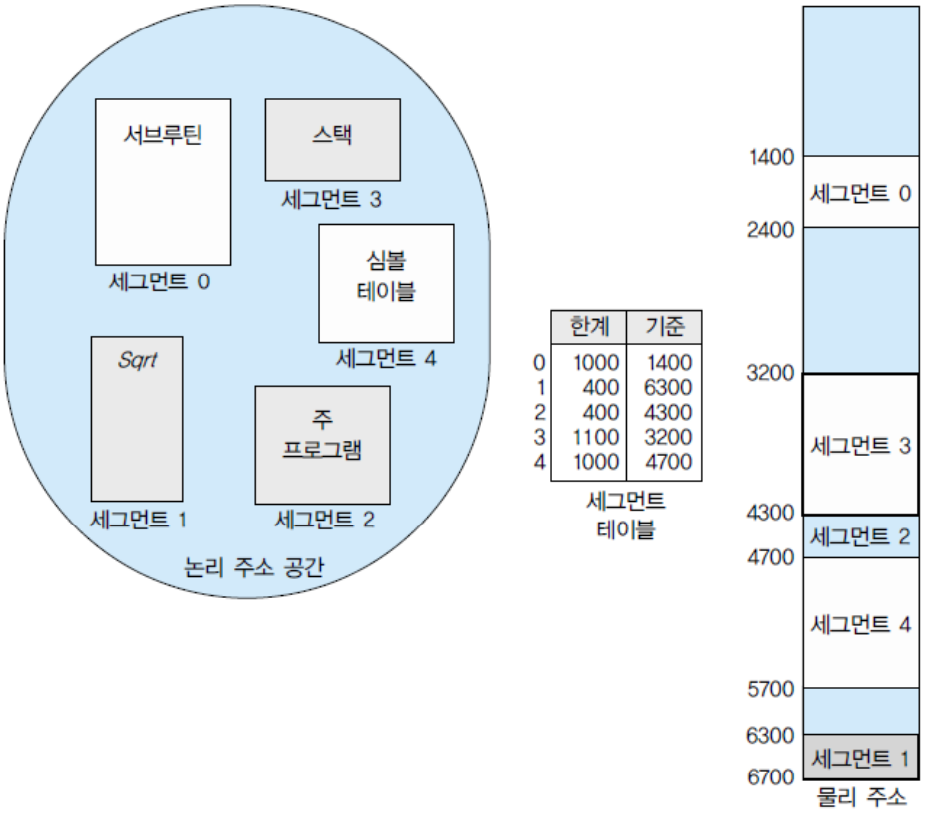


그림 8.20 세그먼테이션의 예

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)

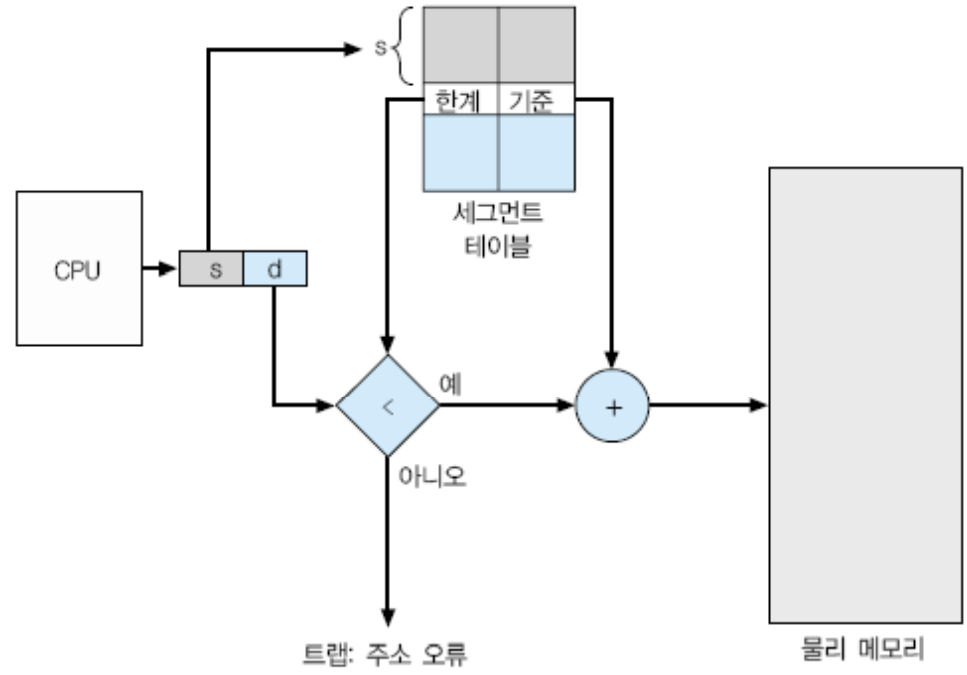


그림 8.19 세그먼테이션 하드웨어

Intel Pentium 사례

- 페이징과 세그먼테이션은 모두 장단점을 가지고 있으며, 어떤 구조(architecture)에서는 둘 다 지원
  - 페이지화된 세그먼테이션(segmentation with paging)
- Pentium 시스템 주소 변환
  - CPU는 논리 주소 생성
  - 세그먼테이션 유니트는 각각의 논리 주소를 선형 주소(linear address)로 변환
  - 페이징 유니트는 선형 주소를 주 메모리의 물리 주소로 변환
  - 세그먼테이션 유니트와 페이징 유니트는 MMU와 동일한 역할

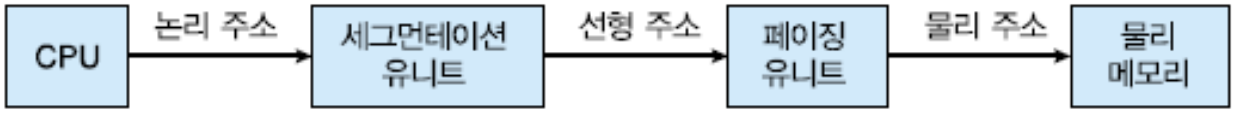


그림 8.21 Pentium에서의 논리 주소에서 물리 주소로의 변환

Pentium 세그먼테이션

- Pentium 구조 세그먼트
  - 하나의 세그먼트가 최대 4 GB의 크기
  - 한 프로세스 당 16 K개의 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

Pentium 논리주소

- 논리 주소는 셀렉터와 변위(selector, offset)의 쌍으로 구성
  - 셀렉터 16비트
  - 변위 32비트
- 셀렉터
  - 다음과 같은 16 비트 수로 구성

  - s는 세그먼트 번호
  - g는 세그먼트가 GDT인지 LDT인지를 표시,
  - p는 보호(protection)와 관련된 정보를 표시
  - 6개의 16비트 세그먼트 레지스터(CS,DS,SS,ES,FS,GS)에 저장
    - 한 프로세스는 최대 6개의 세그먼트를 가리킴

선형주소 변환

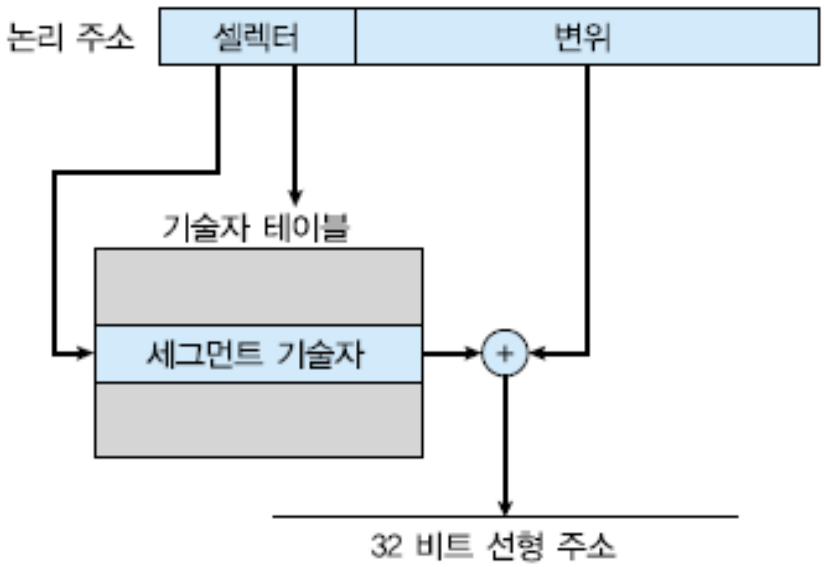
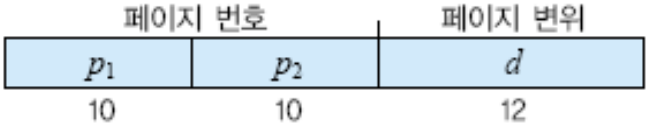


그림 8.22 Intel Pentium 세그먼테이션

Pentium 페이징

- Pentium 구조 페이지
  - 페이지는 4 KB 또는 4 MB
- 2 단계 페이징 기법
  - 4 KB 크기의 페이지를 사용할 때 적용
  - 32 비트의 선형 주소



  - p_1은 페이지 디렉토리(page directory)라고 부르는 최상위 페이지 테이블의 항목을 가리킴
    - CR3레지스터는 현재 프로세스의 페이지 디렉토리를 가리킴
  - p_2는 하위 페이지 테이블의 변위 표시

Pentium 주소변환

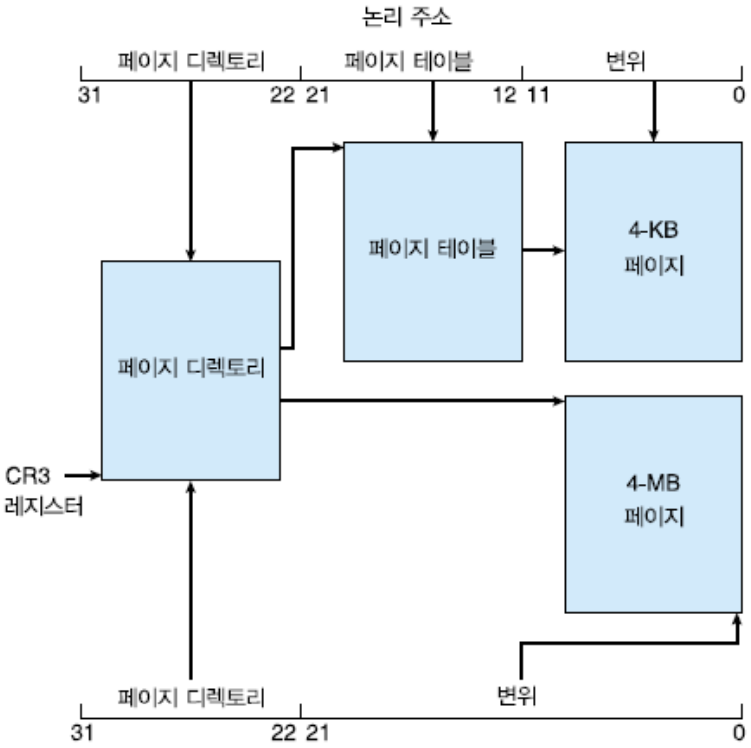


그림 8.23 Pentium 구조의 페이징

Pentium 시스템 기반의 Linux

- Linux는 다양한 처리기에서 동작하도록 최소의 세그먼테이션만 사용
- 6개의 세그먼트를 사용
  - 커널 코드, 커널 자료
  - 사용자 코드, 사용자 자료
  - 작업 상태 세그먼트(task-state segment, TSS)
    - 문맥 교환 시 각 프로세스의 하드웨어 문맥을 저장하는 데 사용
  - 디폴트 지역 기술자 테이블(LDT) 세그먼트
- 사용자 코드와 자료는 모든 프로세스들이 같은 논리 주소 공간을 사용하고 모든 세그먼트 수식자들이 광역 기술자 테이블(GDT)에 저장
- Pentium의 4개 보호모드 중 리눅스는 사용자 모드와 커널 모드 2 개만 사용

Linux 페이징

- 다양한 하드웨어 플랫폼에서 적용 가능하도록 삼 단계 페이지 모델을 사용
  - 선형주소

전역 디렉토리	중간 디렉토리	페이지 테이블	변위 (offset)

  - Pentium 구조는 이 단계 페이지 모델만을 사용
    => Linux는 중간 디렉토리의 크기를 0 비트로 설정하여 실제로 중간 디렉토리를 사용하지 않음

- Linux에서 각 태스크는 각자의 페이지 테이블들을 가짐
  - CR3 레지스터는 현재 수행중인 태스크의 전역 디렉토리(global directory)를 가리킴
  - 문맥 교환 중에 CR3 레지스터의 값은 문맥 작업에 포함된 태스크들의 작업 상태 세그먼트(TSS)에 저장되고 복구

Linux 주소변환

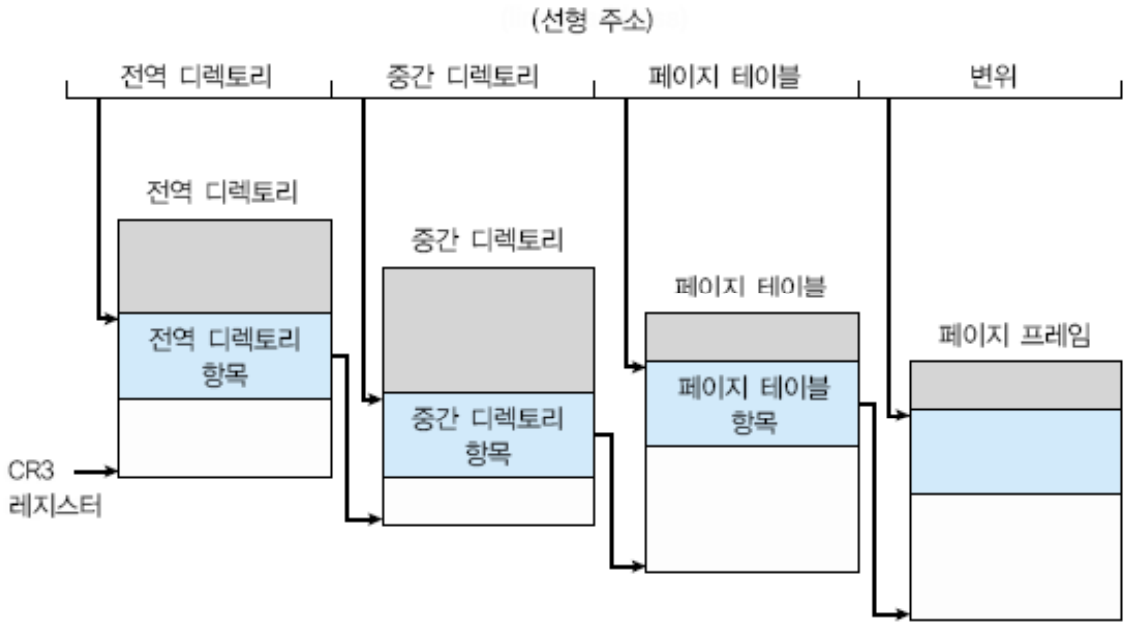


그림 8.24 Linux의 3단계 페이징

연습문제

- 연습문제 8.1
- 연습문제 8.3
- 연습문제 8.9
- 연습문제 8.12