Slide 0

3부 저장장치 관리
(9장. 주기억장치 관리)

배경

Slide 1

- 주소 바인딩 (address binding)
  - 프로그램에서 데이터는 기호주소(변수이름)로 작성
  - 해당 프로그램이 실행되기 전에 실제주소로 바인딩되어야 함
  - 주소 바인딩
    - 기호주소 →컴파일러(compiler) →재배치 가능 주소 (예로 시작위치로부터 14바이트)
    - 재배치 가능 주소 →로더(loader) →절대주소 (예로 74014)
  - 주소바인딩이란 주소 공간 사이의 매핑임

배경 (계속)

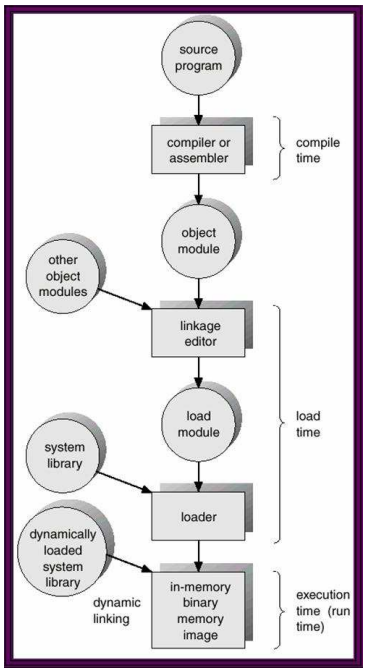
Slide 2

- 주소바인딩 시기
  - 컴파일 시간(compile time)
    - 기억장치내에 적재될 위치가 컴파일 시간에 알 수 있으면 컴파일러는 절대 코드를 생성
    - 예) MS-DOS에서 .COM 형식의 프로그램
    - 적재위치가 바뀌면 다시 컴파일하여야 함
  - 적재시간 (load time)
    - 기억장치내에 적재될 위치를 컴파일 시간에 알 수 없을때는 재배치 코드를 생성
    - 적재될 위치는 적재 시간이 되어야 알 수 있음
    - 주소바인딩은 적재 시간까지 연기됨
    - 적재위치가 바뀌어도 다시 컴파일 할 필요 없음 (재적재 하면 됨)

    ➙ 재배치코드란 기준위치 주소를 사용하는 코드
  - 실행시간 (execution time)
    - 실행 도중에 적재 위치가 바뀔 수 있다면 바인딩은 실행시에 이루어짐
    - 대부분의 범용 운영체제가 사용하는 방법

배경 (계속)

Slide 3



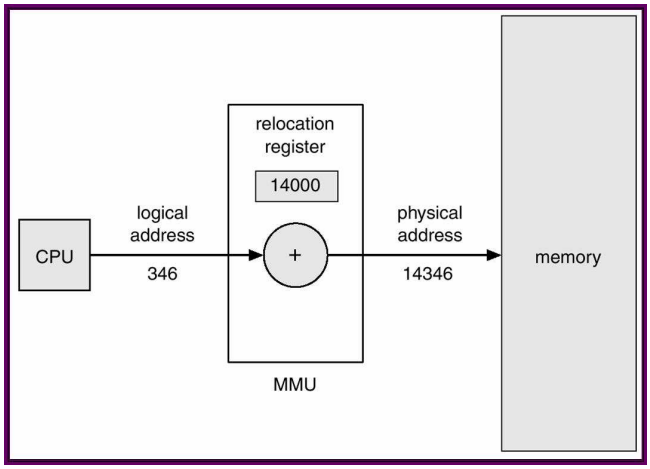
배경 (계속)

Slide 4

- 논리적/물리적 주소 공간
  - 논리주소
    - CPU가 생성하는 주소
  - 물리주소
    - 주기억 장치에 가해지는 주소
  - 주소바인딩과 논리/물리 주소
    - 논리주소 = 물리주소
      - 컴파일 시간 바인딩
      - 적재시간 바인딩
    - 논리주소 ≠ 물리주소
      - 실행시간 바인딩
      - 이때 논리주소를 가상주소 (virtual address)라 함
      - 논리주소 →물리주소 매핑은 MMU (Memory Management Unit)가 담당
      - 간단한 매핑 방법: 재배치 레지스터 (relocation register)를 사용한 방법

배경 (계속)

Slide 5



  - 주소공간
    - 논리적 주소공간: 프로그램에의해 참조될 수 있는 모든 논리적 주소
    - 물리적 주소공간: 논리적 주소공간에 대응되는 모든 물리적 주소 (다른말로, 주기억장치 주소공간)

배경 (계속)

Slide 6

- 동적 적재(Dynamic Loading)
  - 적재방법
    - 전체 프로그램을 모두 주기억장치에 적재하는 방법
      - 프로세스의 크기가 물리적 주기억장치의 크기보다 작아야 함
      - 실제 실행되지 않는 부분도 주기억장치에 적재 됨
    - 루틴이 필요할 때 해당 루틴을 적재하는 방법 (동적 적재)
      - main 프로그램이 적재되고
      - 다른 루틴은 필요시 적재함 (relocatable linking loader에 의하여)
      - 사용되지 않는 루틴은 적재되지 않음
        (예로서 많은 양의 오류처리 루틴)

배경 (계속)

Slide 7

- 동적 연결 (dynamic linking) 과 공유라이브러리 (shared libraries)
  - 프로그램과 라이브러리 연결방법
    - 정적 연결 (static linking)
      - 프로그램과 라이브러리가 결합되어 실행파일을 만듦
      - 모든 프로그램이 라이브러리의 복사본을 가짐
      - 실행파일 크기가 커지고 주기억장치를 많이 사용 (낭비)
    - 동적 연결
      - 프로그램과 라이브러리가 결합되지 않고 실행시 연결 됨
      - 프로그램에 스터브 (stub)가 포함됨

배경 (계속)

Slide 8

- 스터브
  - 적은 양의 코드
  - 주기억장치에 공유되는 라이브러리 루틴에서 필요한 루틴을 찾아 줌
  - 루틴이 없을 경우 라이브러리를 적재하는 작업을 수행
  - 루틴을 처음 호출시에만 스터브가 사용되며 재 호출시는 주소정보를 이용하여 바로 호출
- 동적 연결의 장점
  - 라이브러리 갱신이 용이
  - 여러 버전의 라이브러리 사용가능 (각 프로그램 별로 자신에게 적절한 라이브러리 버전을 사용)
  - 디스크공간 주기억장치 공간을 절약

배경 (계속)

Slide 9

- 중첩 (Overlays)
  - 설명
    - 주기억장치 공간을 동시에 사용되지 않는 두개의 코드가 시간상으로 나누어 사용
    - 프로세스에게 할당된 공간보다 프로세스가 큰 경우에 유용
    - 결과적으로 적은 주기억장치를 효율적으로 사용할 수 있음
    - 주기억장치 크기가 적거나 주기억장치를 위한 진보된 하드웨어 지원이 부족한 경우 사용
  - 예) 두 단계 어셈블러

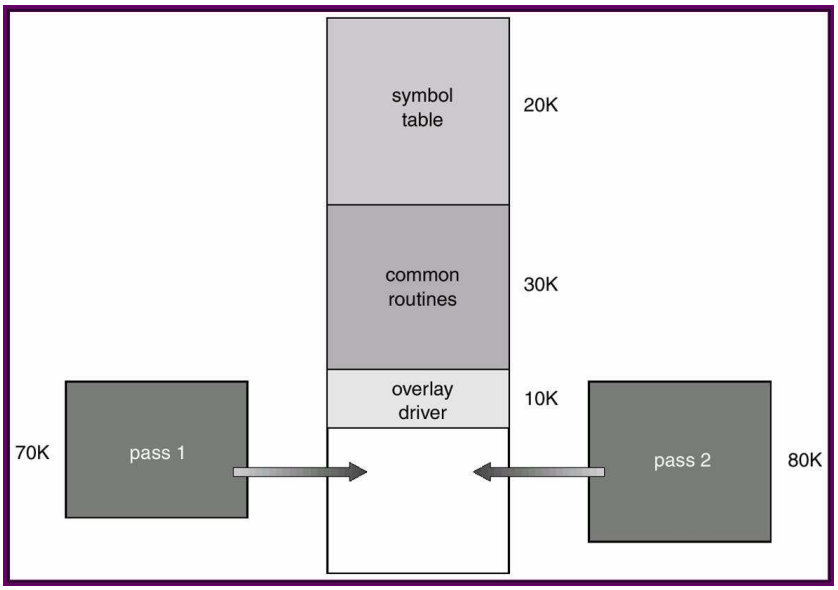
패스 1 코드	70KB
패스 2 코드	80KB
심볼 테이블	20KB
공통 루틴	30KB

    - 중첩
      - 중첩 A: 패스 1 코드 + 심볼 테이블 + 공통루틴
      - 중첩 B: 패스 2 코드 + 심볼 테이블 + 공통루틴

배경 (계속)

Slide 10

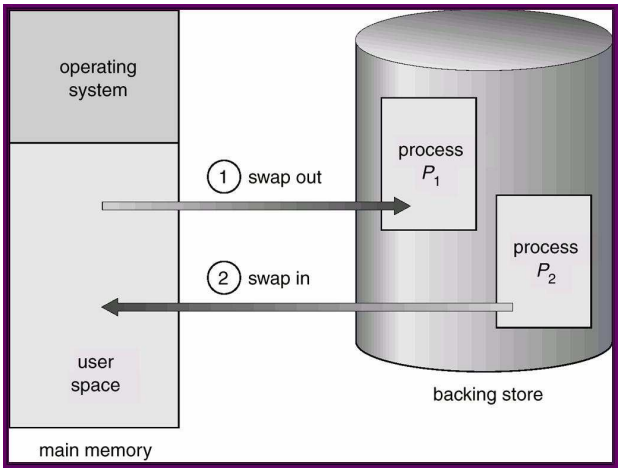
* 중첩 드라이버 (overlay driver)를 사용



스왑핑 (Swapping)

Slide 11

- 스왑핑이란?
  - 주기억장치 용량 < 실행중인 프로세스들의 전체 크기
  - 프로세스를 일시적으로 디시크로 옮기고 수행시 다시 주기억장치로 옮기는 과정



스왑핑 (Swapping) (계속)

Slide 12

- 롤인(roll-in) 롤아웃(roll-out)
  - 우선순위에 근거한 스왑핑을 말함
  - 프로세스 수행 중 우선순위가 높은 프로세스가 들어오면 현재 프로세스를 스왑아웃 시킴
- 스왑핑시 적재 주소
  - 컴파일시간 바인딩 혹은 적재 시간 바인딩의 경우
    * 스왑아웃 되었다가 스왑인 될 때 같은 주소로 적재되어야함
  - 실행시간 바인딩
    * 스왑인 될 때 적재 주소가 결정됨

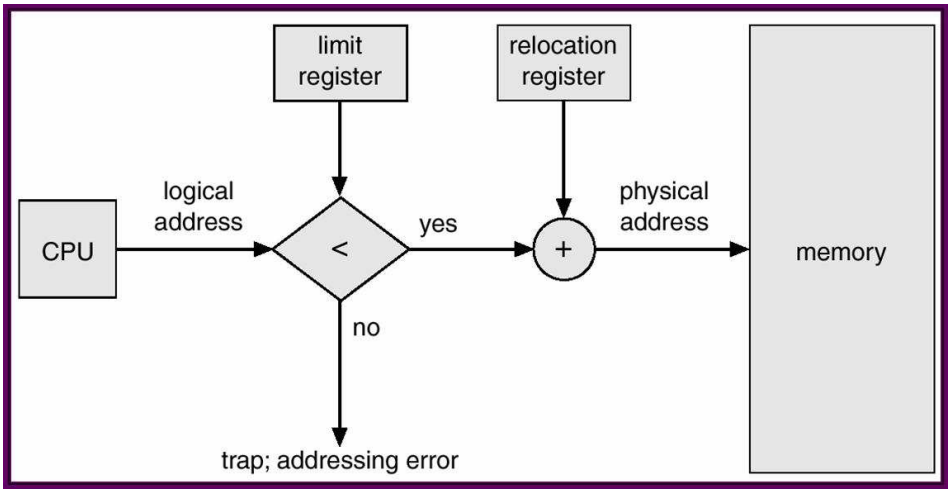
연속공간 할당

Slide 13

- 주기억장치 할당
  - 상주 운영체제와 사용자 프로세스들이 주기억장치에 적재됨
  - 이때 각 프로세스를 연속된 단일 공간에 할당함
  - 보통 상주 운영체제는 인터럽트 벡터테이블이 있는 하위메모리에 사용자 프로세스는 상위메모리에 할당함
- 주기억장치 보호
  - 운영체제와 사용자프로세스간 그리고 사용자 프로세스간 보호해야함
  - 주기억장치 보호를 위해 재배치 레지스터와 한계레지스터를 사용 (참고, 2장에서는 재배치 레지스터가 기준레지스터라 불렸음)
  - 재배치 레지스터 $\leq$ CPU 주소 $<$ (재배치 레지스터 + 한계 레지스터)

연속공간 할당 (계속)

Slide 14



연속공간 할당 (계속)

Slide 15

- 주기억장치 할당
  - 두 방법
    - MFT (Multiprogramming with Fixed number of Tasks)
    - MVT (Multiprogramming with Variable number of Tasks)
  - MFT
    - 주기억장치를 고정된 크기의 영역으로 분할
    - 각 영역을 하나의 프로세스에게 할당
    - 다중 프로그래밍 정도는 분할 영역수에의해 결정됨
    - 각 영역의 사용여부를 테이블로 관리
    - IBM OS/360에서 사용한 방법으로 현재는 사용 안함
  - MVT
    - 사용공간과 미사용 공간으로 구분하여 테이블로 관리
    - 처음에는 운영체제공간 이외에 공간이 모두 사용가능 (홀(hole) 이라함)
    - 프로세스가 도착하면 그 프로세스를 할당할 수 있는 홀을 찾아 할당
    - 시간이 지날수록 전반적으로 홀의 크기가 작아지며 흩어짐

연속공간 할당 (계속)

Slide 16

- MVT에서 동적 기억장치 할당
  * 여러 개의 홀중 어떤 홀을 할당할 것인가를 결정하는 방법
  * 3가지 방법
    · 최초적합(first-fit): 첫 번째로 할당할 수 있는 홀에 할당
    · 최적적합(best-fit): 할당할 수 있는 가장 작은 홀에 할당
    · 최악적합(worst-fit): 가장 큰 홀에 할당
  * 외부 단편화 (external fragmentation) 문제 발생
    · 홀 전체의 크기로는 만족하나 홀들이 연속적이지 않아 프로세스를 할당할 수 없는 상황
    · compaction으로 이 문제를 해결
    · compaction이란 프로세스 적재 위치를 재배치하여 흩어져 있는 홀을 큰 하나의 홀로 만드는 것을 말함
  * 내부 단편화 (internal fragmentation)
    · 18464의 홀에 18462의 프로세서를 할당시
      ➡ 2바이트의 홀을 남기지 않고 해당 프로세스에게 모두 할당함
      ➡ 이는 적은 바이트의 홀을 관리하는 것이 오히려 비용이 많이 소요됨으로
    · 이때 발생하는 2바이트를 내부 단편화라 함 (이 공간은 사용되지 못함)

연속공간 할당 (계속)

Slide 17

* 외부 단편화 와 내부 단편화 정리
  · 외부 단편화
    ➡ 프로세스에게 할당되지 않았으나
    ➡ 홀이 프로세스 크기보다 작아서 빈 공간으로 있는 것
    ➡ 홀보다 작은 크기의 프로세스나 혹은 compaction으로 빈공간 사용 가능
  · 내부 단편화
    ➡ 프로세스에게 할당하였으나
    ➡ 프로세스 크기가 할당보다 작아서 빈 공간으로 있는 것
    ➡ 빈 공간은 해당 프로세스가 점유하고 있는 동안에는 사용될 수 없음

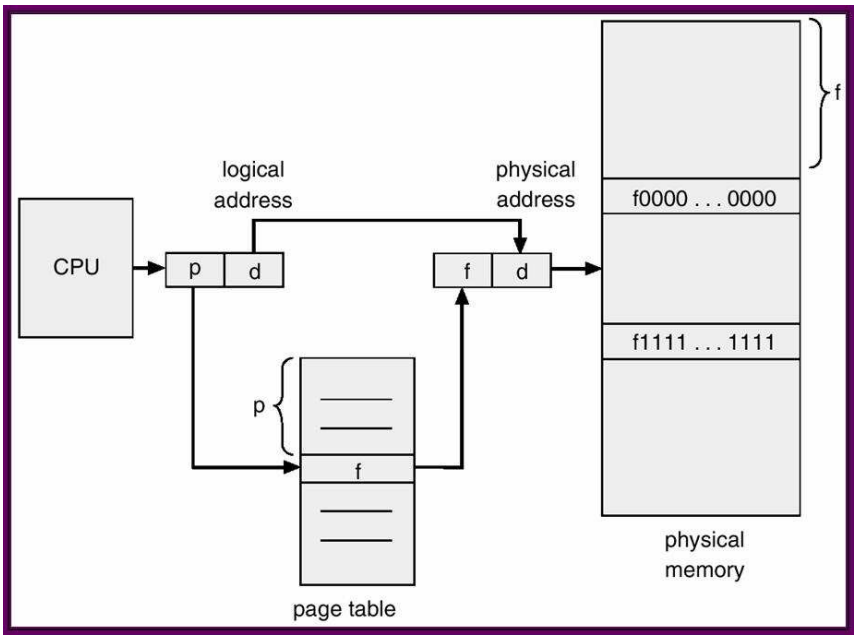
페이징(Paging)

Slide 18

- 페이징이란?
  - 외부 단편화 문제를 해결하는 한 방법
  - 현재 가장 많이 사용하는 방법으로 기억장치를 일정한 단위로 분할
- 기억장치의 분할
  - 논리적 기억장치를 페이지(page) 단위로 나눔
  - 물리적 기억장치를 프레임(frame) 단위로 나눔
  - 페이지와 프레임은 같은 크기
    - 보통, 512, 1KB, ..., 8KB까지 사용
  - 논리주소를 물리주소로 바꾸어 주기억장치 액세스 (하드웨어 지원하에)
    - 논리주소(CPU 주소): 페이지 번호(p) + 변위(d)
    - 페이지 번호는 페이지 테이블의 색인으로 사용
    - 페이지 테이블: 페이지 번호 →물리적 기억장치의 기준주소
    - 물리주소(주기억장치 주소): 기준주소(f) + 변위(d)

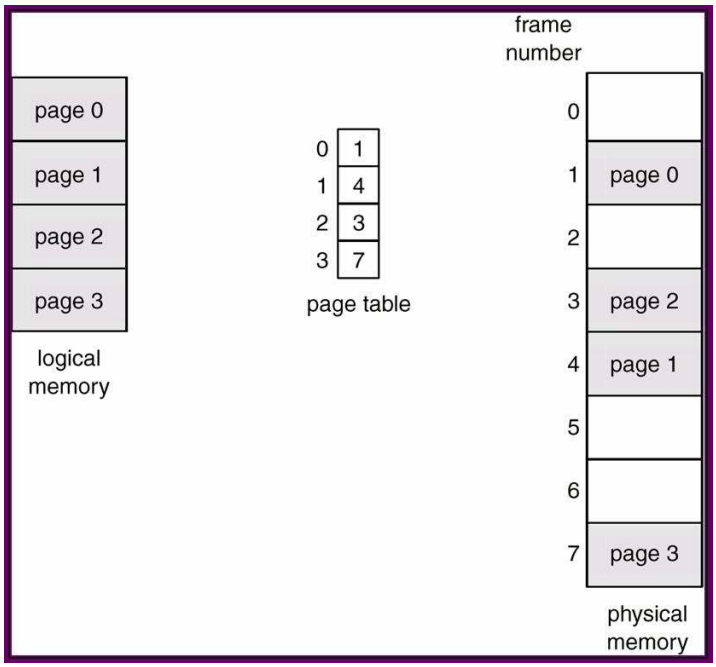
페이징(Paging) (계속)

Slide 19



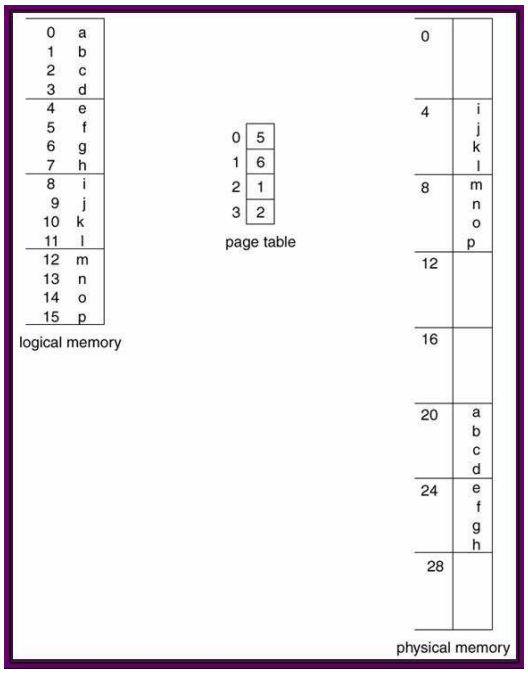
페이징(Paging) (계속)

Slide 20



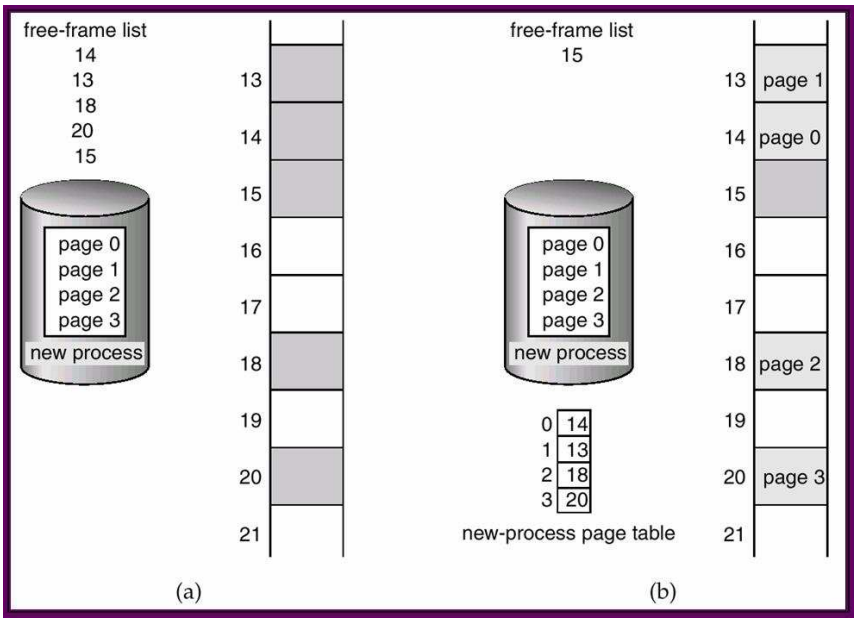
페이징(Paging) (계속)

Slide 21



페이징(Paging) (계속)

Slide 22



페이징(Paging) (계속)

Slide 23

- 페이징 사용시
  - 외부 단편화는 발생하지 않음
  - 내부 단편화는 발생
    - 프로세스의 마지막 페이지는 프레임크기보다 작을 수 있음
    - 최악의 경우: n 페이지 + one 워드
    - 보통 프로세스별로 반 프레임 정도 발생
    - 페이지 크기를 줄임 →내부 단편화 작아짐 →페이지 테이블 커짐
- 운영체제
  - 프레임 테이블을 이용하여 프레임 할당내역을 관리하여야함
  - 즉, 프레임별 할당여부, 할당되었을 경우 어떤 프로세스의 몇 번 페이지인지등
  - 각 프로세스별로 PCB내에 페이지 테이블 관리
  - 페이징은 문맥전환에 소요되는 시간을 증가 시킴

페이징(Paging) (계속)

Slide 24

- 하드웨어 지원
  - 페이지 테이블의 하드웨어 구현 방법
    * 전용 레지스터 집합을 사용
      · 페이지 테이블이 작은 경우에만 가능
      · 보통 256항목 정도
    * 주기억장치 사용
      · 주기억장치의 특정 영역에 저장
      · 이 테이블의 주소를 PTBR (page-table base register)에 저장
      · 문제점: 항상 두번 (테이블과 데이터) 접속해야함 →속도가 느려짐

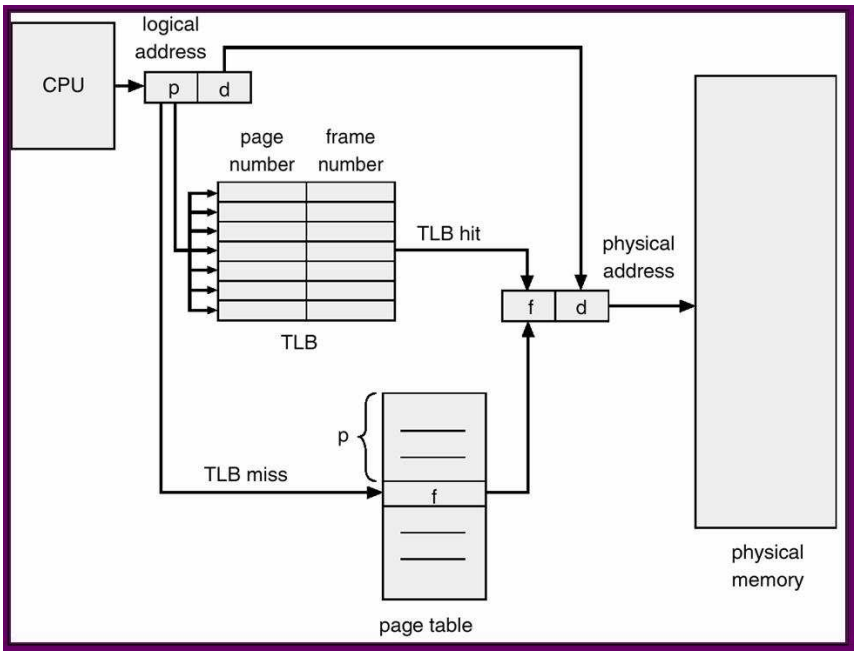
페이징(Paging) (계속)

Slide 25

  - 두번 접속 문제 해결 →전용 캐시를 사용
    * 이 캐시를 TLB(Translation Look-aside Buffer)라 함
    * 연관매핑 (associative mapping) 사용
    * 탐색은 빠르나 크기가 작고 (8 ∼2048 항목) 비용이 비쌈
    * TLB 검색후 없으면 주기억장치 참조
    * TLB는 문맥교환에 의하여 전체가 삭제되어야함
    * TLB에는 제거될 수 없는 정보가 있음 (이를 wired down 되었다함)
      →보통 커널 페이지등이 이에 해당
    * TLB에 ASID(Address-Space IDentifier)를 저장하는 것도 있음
      · 프로세스별로 유일하게 주어짐
      · 프로세스간의 주기억장치 보호에 사용될 수 있음
      · 여러 프로세스의 페이지가 동시에 TLB에 존재할 수 있음
    * Effective Access Time (EAT)
      · 주기억장치 접속: $1\mu s$
      · TLB lookup: ϵ
      · TLB hit ratio: α
      · EAT $= (1+\epsilon)\alpha + (2+\epsilon)(1-\alpha) = 2+\epsilon-\alpha$

페이징(Paging) (계속)

Slide 26



페이징(Paging) (계속)

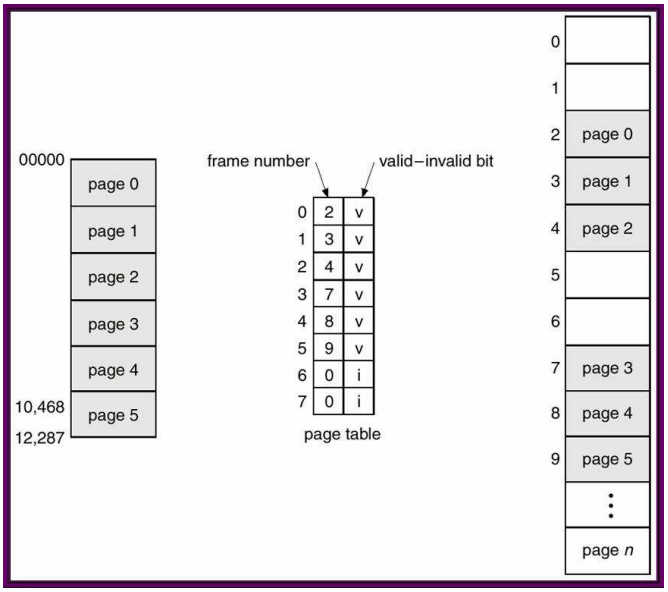
Slide 27

- 보호
  - 페이징 기법에서 주기억장치 보호 방법
  - 페이지 테이블에 보호비트를 사용
  - 보호비트의 예
    * 읽기전용/읽기-쓰기 비트
      · 읽기-쓰기 페이지인지 읽기전용 페이지인지를 나타냄
      · 읽기전용 페이지에대한 쓰기를 막음
      · 불법적인 시도 (읽기전용에 쓰기등) 트랩(trap)을 발생시킴
    * 유효/무효 비트
      · 해당 페이지가 논리주소공간에 존재하는 페이지인지 및 적합한 페이지인지를 나타냄
      · 유효/무효 비트는 페이지별로 존재해 많은 기억공간이 소요됨
      ➙ 페이지 테이블 길이 레지스터 (PTLR: page table length register)를 대신 사용하기도 함

페이징(Paging) (계속)

Slide 28

· 유효/무효 비트 예)



페이징(Paging) (계속)

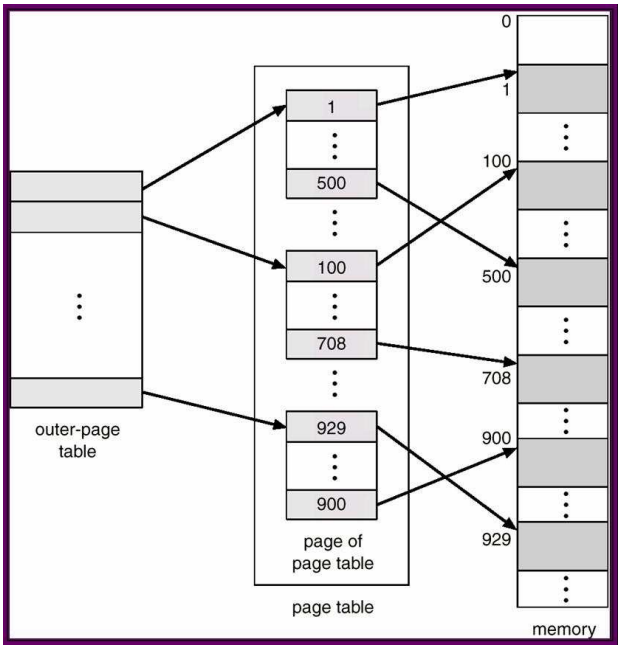
Slide 29

- 페이지 테이블의 구조
  - 계층구조 페이징
    - 현대 컴퓨터는 매우 큰 논리주소 공간을 가짐: ($2^{32} \sim 2^{64}$)
    - 한 페이지 크기가 4KB (2^{12}) 이면 페이지 테이블의 항목은 1M (2^{20}) 필요
    - 각 항이 4B (32비트) 이면 한 프로세스의 페이지 테이블 저장은 4MB 필요
    - 4MB를 연속적인 주소공간에 저장하는 것은 어려움
    - ⇨ 페이지 테이블을 페이징하는 기법 (계층구조 페이징)
    - 예) 두 단계 페이징

page number		page offset
p_1	p_2	d
10	10	12

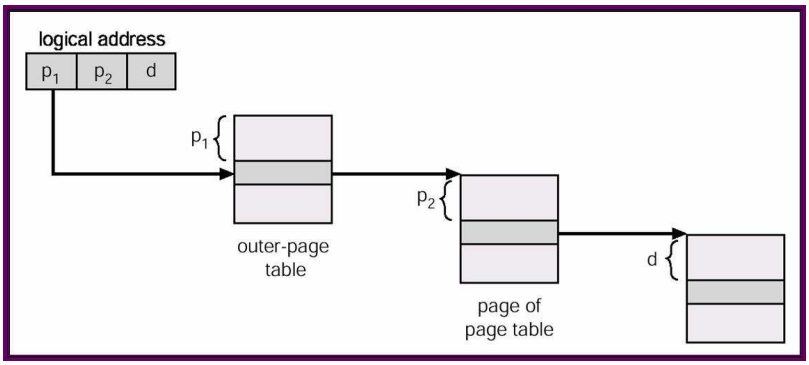
페이징(Paging) (계속)

Slide 30



페이징(Paging) (계속)

Slide 31



* 64비트 논리주소 공간을 갖는 시스템
  - 두 단계 페이징도 부족 p_1이 42비트 (2^{42} 이면 4TB) 필요
  - 다 단계 페이징 방식을 사용
  - 단계가 많아질수록 더 많은 시간이 소요됨

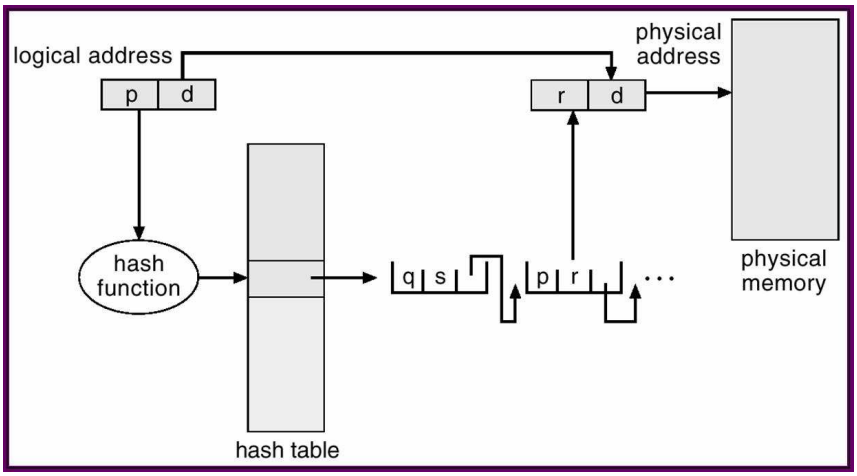
페이징(Paging) (계속)

Slide 32

- 해시된 페이지 테이블
  - 다 단계 페이징에서 단계가 많아질수록 많은 시간이 소요되는 문제를 보완
  - 보통 32비트보다 큰 주소공간에서 사용
  - 이 테이블을 해시 페이지 테이블로 부름
  - 테이블 구성
    - 가상 페이지 번호
    - 페이지 프레임 값
    - 다음 요소를 가리키는 포인터
  - 동작
    - 가상 페이지 번호를 해시하여 해시 테이블의 위치를 계산
    - 해당 위치에 있는 연결 리스트를 검색하여 프레임 값을 얻음
    - 예)

페이징(Paging) (계속)

Slide 33



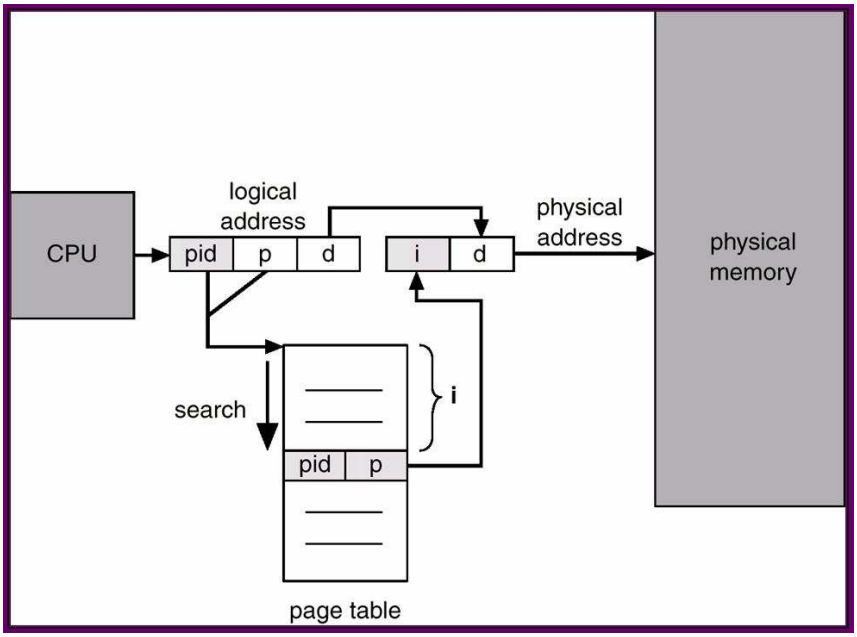
페이징(Paging) (계속)

Slide 34

- 역 페이지 테이블
  - 페이지 테이블 사용의 문제점
    - 각 프로세스별로 별도로 존재
    - 테이블 크기가 매우 큼
      페이지주소 20비트 사용한다면 $\rightarrow 2^{20}$ 항목에 4B (4MB)
    - 이에대한 해결책으로 역 페이지 테이블 사용
  - 역 페이지 테이블이란
    - 프로세스별 페이지번호에 대한 프레임번호 매핑이 아니라
      주기억장치 프레임에 대한 프로세스 및 페이지번호로 매핑함
    - 주기억장치 액세스
      - 논리주소: 페이지번호 + 변위
      - 프로세스 식별자와 페이지번호를 이용하여 역 페이지 테이블 탐색
      - 일치하는 것이 발견되면 그 항목의 프레임에 변위를 더해서 물리적주소 생성
    - 문제
      - 역 페이지 검색에 많은 시간 소요됨 $\rightarrow$해시 테이블 활용

페이징(Paging) (계속)

Slide 35



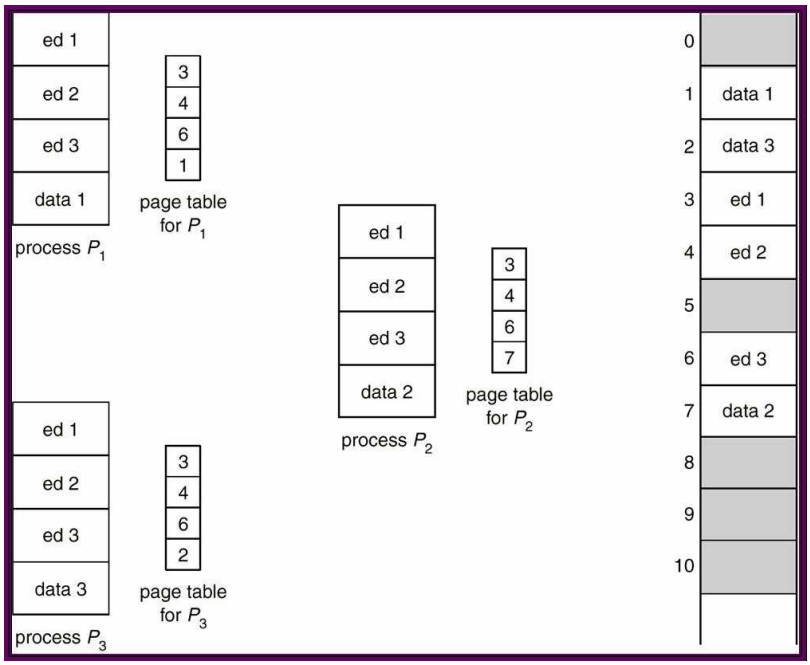
페이징(Paging) (계속)

Slide 36

- 공유페이지
  - 공유페이지란
    - 여러 프로세스가 공유하는 페이지
    - 페이징의 한 장점
  - 페이지가 공유되기 위한 조건
    - 해당 페이지가 재진입 코드여야함
    - 재진입 (혹은 pure code) 코드란
      - 스스로 수정하지 않는 코드
      - 즉, 수행 중 변경되지 않는 코드
    - 역 페이지 테이블 방식에서는 페이지 공유가 어려움

페이징(Paging) (계속)

Slide 37



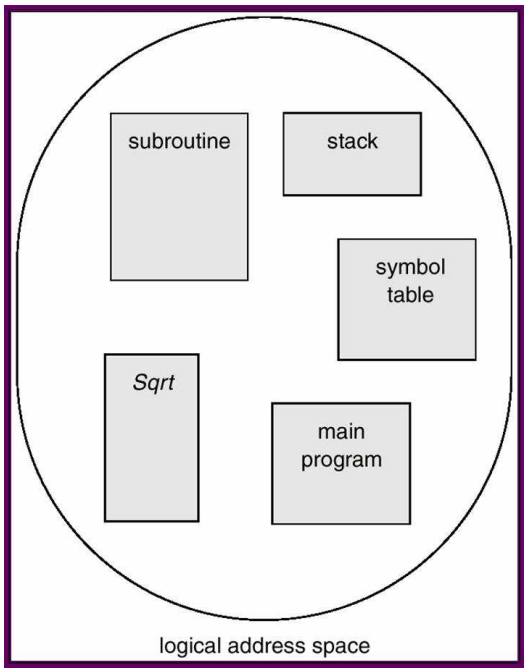
세그먼테이션

Slide 38

- 세그먼테이션이란?
  - 보통 사용자는 프로그램을 세그먼트의 모음으로 생각
  - 세그먼트란 논리적 단위로서 서브루틴, 프로시저, 함수, 모듈, 테이블, 스택등을 말함
  - 세그먼테이션은 이러한 관점을 지원해 주는 주기억장치 관리 기법
  - 사용자는 논리주소공간을 세그먼트의 집합으로 간주
  - 세그먼트: 세그먼트이름 + 길이
  - 주소지정: 세그먼트이름 + 변위

세그먼테이션 (계속)

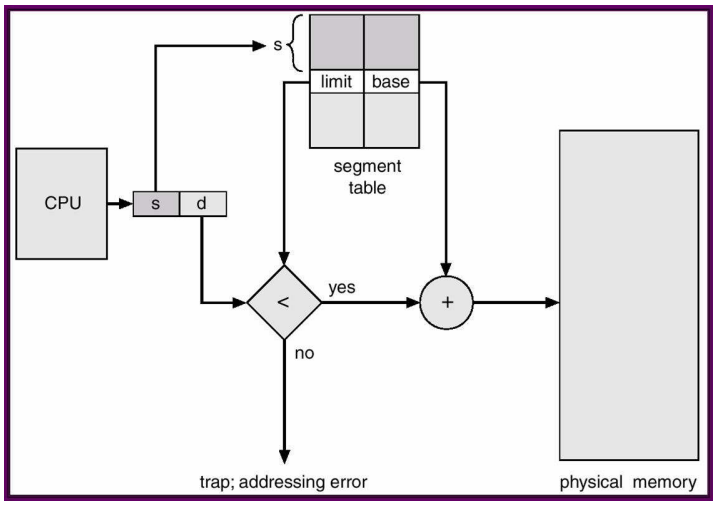
Slide 39



세그먼테이션 (계속)

Slide 40

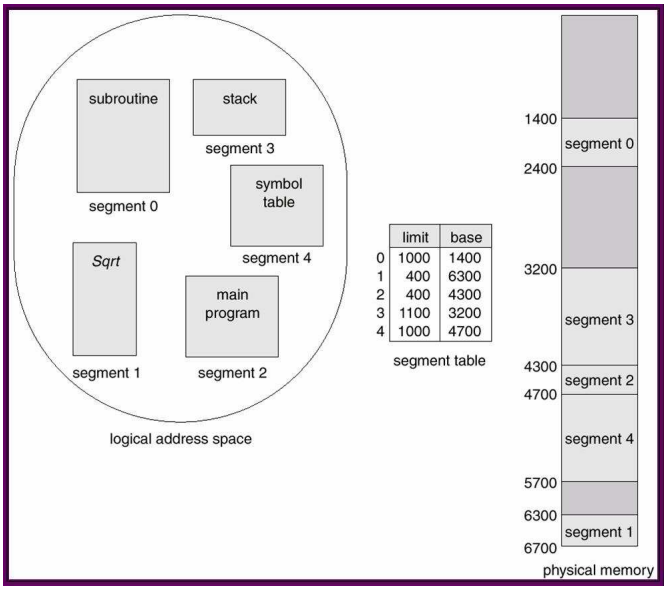
- 하드웨어
  - 세그먼트 주소 →물리적 주소 변환은 세그먼트 테이블을 이용
  - 세그먼트 테이블 항목: 기준(base) + 한계(limit)



세그먼테이션 (계속)

Slide 41

- 세그먼테이션의 예)



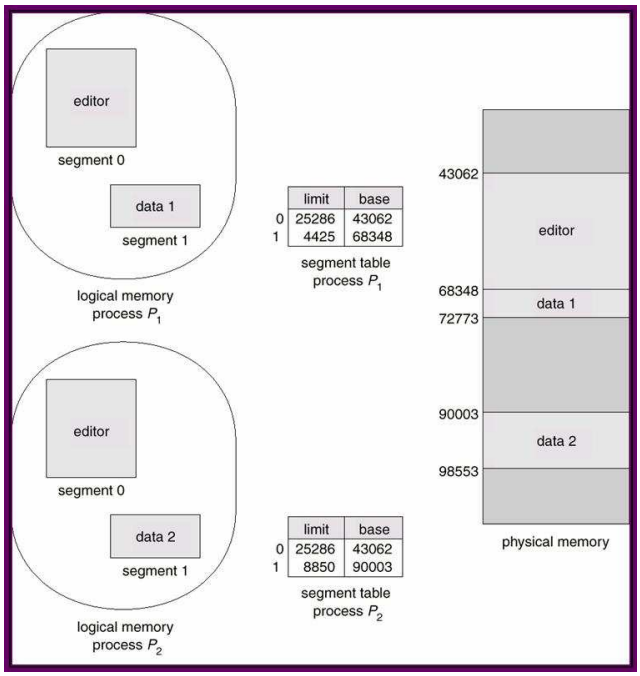
세그먼테이션 (계속)

Slide 42

- 보호와 공유
  - 세그먼트는 보호에 유리함
    - 세그먼트 별로 성격(읽기-전용(코드) 혹은 실행-전용등)이 명확함으로
    - 세그먼트 별로 보호비트 지정 가능함 (페이징은 페이지별로 지정)
  - 세그먼트는 공유에 유리함
    - 페이징이 공유를 페이지 단위로 하는데 반하여 세그먼트로 단위로 가능함
    - 단, 공유되는 세그먼트는 모든 프로세들에게서 같은 세그먼트 번호를 갖도록 해야함
- 단편화
  - 페이징이 내부단편화 발생하는데 반해 외부 단편화가 발생함

세그먼테이션 (계속)

Slide 43



페이징과 세그먼테이션의 결합

Slide 44

- 인텔 80386
  - 설명
    - 각 프로세스는 최대 16K개의 세그먼트를 가질 수 있음
    - 각 세그먼트는 최대 4GB 크기가 가능
    - 각 세그먼트는 페이징됨
    - 페이지의 크기는 4KB
  - 한 프로세스의 논리공간은 크게 두 영역으로 분할
    - 영역1
      - 프로세스가 독점적으로 사용하는 세그먼트로 최대 8K개 가능
      - LDT (Local Descriptor Table)에 저장됨
    - 영역2
      - 다른 프로세스와 공유하는 세그먼트로 최대 8K개 가능
      - GDT (Global Descriptor Table)에 저장됨
    - LDT, GDT의 각 항은 8바이트로 세그먼트의
      - 기준
      - 한계
      - 액세스 권리 정보가 있음

페이징과 세그먼테이션의 결합 (계속)

Slide 45

  - 주소변환
    - 논리주소: 선택자 (16비트) + 오프셋 (32비트)
    - 선택자: 세그먼트 번호 (13비트) + GDT/LDT (1비트) + 보호 (2비트)
      - 선택자는 6개의 16비트 세그먼트 레지스터에 저장됨
      - 세그먼트 레지스터의 세그먼트 번호로 LDT/GDT의 8바이트 읽음
      - LDT/GDT 8바이트 항목중 기준, 한계, 권리를 이용 오류 점검
    - LDT/GDT의 기준주소 (32비트) + 오프셋 (32비트) →선형주소 (linear address) 생성
    - 선형주소 페이징
      - 2단계 페이징: p_1(10비트) + p_2(10비트) + d(12비트)
      - 물리주소 생성

페이징과 세그먼테이션의 결합 (계속)

Slide 46