운영체제: 강의노트 10

A. Silberschatz, P.B. Galvin, G. Gagne
Operating System Concepts,
Sixth Edition, John Wiley & Sons, 2003.

Part III. Storage Management

9 주기억장치 관리

9.1 배경

9.1.1 주소 바인딩

- 프로그램을 작성할 때, 프로그래머는 기호 주소를 사용한다. 즉, 변수를 선언하고 이 변수 이름을 통해 주소를 접근한다.
- 프로그램은 실행되기 전에 이런 기호 주소를 실제 주소로 바인딩해주어야 한다.
- 주소를 바인딩하는 시기
  - 컴파일 시간: 컴파일러가 프로세스가 어디에 적재될지 알고 있으면 컴파일할 때 주소를 바인딩할 수 있다.
  - 적재 시간: 컴파일할 때 프로세스가 어디에 적재될지 모르면 컴파일러는 **재배치 가능 코드(relocatable code)**를 생성한다. 실제 바인딩은 적재할 때까지 지연된다. 재배치 가능 코드란 기준 위치 주소를 사용하는 코드를 말한다.
  - 실행 시간: 실행 도중에 프로세스의 적재 위치가 바뀔 수 있으면 바인딩은 실행될 때 이루어진다. 현재는 대부분 이 방법을 사용한다.

9.1.2 논리 VS. 물리적 주소 공간

- CPU에 의해 생성되는 주소는 보통 논리적 주소라 하고, 주기억장치가 접하게 되는 주소(MAR에 적재되는 주소)는 물리적 주소라 한다.
- 컴파일 시간 바인딩이나 적재 시간 바인딩에서는 논리적 주소와 물리적 주소가 같다.
- 실행 시간 바인딩의 경우에는 논리적 주소와 물리적 주소가 다르며, 이 경우에 논리적 주소를 가상 주소(virtual address)라 한다.
- 프로그램에 의해 참조될 수 있는 모든 논리적 주소를 논리적 주소 공간(logical address space)이라 한다. 모든 논리적 주소에 대응되는 모든 물리적 주소를 물리적 주소 공간이라 한다.
- 실행 시간 바인딩에서 논리적 주소와 물리적 주소 간에 매핑은 주기억장치-관리 장치(MMU, Memory-Management Unit)라 하는 하드웨어에 의해 이루어진다.
- 간단한 예) 재배치 레지스터(relocation register) 사용
  - 재배치 레지스터는 다른 말로 기저 레지스터(base register)라 하며, 이 레지스터에 있는 값이 b이고 CPU가 생성한 논리적 주소가 a이면 주기억장치에게는 $a + b$ 값이 주소 정보로 전달된다.
  - MSDOS는 네 개의 기저 레지스터를 사용하였다.

9.1.3 동적 적재

- 초창기에는 프로그램이 실행되기 위해서는 전체 프로그램이 모두 주기억장치에 적재되어야 했다.
  - 문제점: 프로세스의 크기가 물리적 주기억장치 크기에 의해 제한된다.
- **동적 적재(dynamic loading)**: 루틴이 호출될 때 주기억장치에 적재하는 방법
  - 모든 루틴은 재배치 가능 코드 형태로 디스크에 유지된다.
  - main 프로그램이 먼저 적재되며, 실행 도중에 다른 루틴을 호출해야 하면 이 루틴이 현재 주기억장치에 있는지 검사하고 없으면 재배치 가능 연결 적재기(relocatable linking loader)를 이용하여 루틴을 주기억장치에 적재한다.
  - 장점: 사용하지 않는 루틴(오류 처리 루틴)은 주기억장치에 적재되지 않는다.

9.1.4 동적 연결과 공유 라이브러리

- **정적 연결(static linking)**: 프로그래밍 언어에서 제공하는 라이브러리를 프로그램과 결합하여 사용하는 방식
  - 단점: 모든 프로그램은 라이브러리의 복사본을 가지고 있다. 따라서 디스크 공간과 주기억장치 공간이 모두 낭비된다.
- **동적 연결(dynamic linking)**: 라이브러리와의 연결이 실행 시간까지 연기되는 방식
  - 동적 연결의 경우 각 프로그램에 stub가 포함되며, 이 stub는 주기억장치에 공유되고 있는 라이브러리 루틴에서 프로그램이 필요하는 루틴을 찾아주는 적은 양의 코드이다.
  - 어떤 루틴을 처음 호출할 때만 stub가 필요하며, 그 루틴을 다시 호출하면 주소 정보를 이용하여 바로 호출한다.

  - 장점
    - 라이브러리 갱신이 용이하다.
    - 여러 버전의 라이브러리를 사용할 수 있다. 이 경우에 각 프로그램은 자신의 버전 정보를 이용하여 적절한 라이브러리를 사용하게 된다.
    - 디스크 공간과 주기억장치 공간을 절약할 수 있다.

- 동적 연결은 다른 주소 공간을 접근해야 하므로 운영체제 지원이 필요하다.

9.1.5 중첩

- **중첩(overlay)**은 프로세스의 크기가 프로세스에게 할당된 공간보다 클 수 있도록 해준다. 기본 생각은 현재 필요한 명령어들과 데이터만 주기억장치에 유지하는 것이다.

- 예) 두 단계 어셈블러(2-pass assembler)란 어셈블리로 작성된 프로그램을 두 단계로 나누어 번역하는 어셈블러를 말한다. 이 어셈블러의 구성요소는 다음과 같다.

Pass 1	70 KB
Pass 2	80 KB
기호 테이블	20 KB
공통 루틴	30 KB

  기호 테이블과 공통 루틴은 Pass 1과 Pass 2에서 모두 필요하다. 모든 구성요소를 모두 적재하면 200 KB가 필요하다. 하지만 중첩을 사용하면 Pass 1을 실행할 때에는 120 KB만 필요하고, Pass 2를 실행할 때에는 130 KB만 필요하다.

- 중첩은 운영체제 지원이 필요없지만 프로그래머는 중첩을 고려하여 프로그램을 작성해야 하므로 현재는 사용하지 않는 방법이다.

9.2 스왑핑

- 프로세스는 실행 도중에 일시적으로 주기억장치에서 디스크로 옮겨진 후에 나중에 다시 주기억장치에 적재되어 실행을 재개할 수 있다. 이 과정을 **스왑핑(swapping)**이라 한다.

- 스왑핑이 우선순위에 의해 발생되면 roll out, roll in이라는 용어를 사용한다.

- 컴파일 시간 바인딩 또는 적재 시간 바인딩을 사용하는 경우에는 스왑아웃(swap out)되었다가 스왑인(swap in) 되었을 때 같은 물리적 주소로 적재되어야 한다. 하지만 실행 시간 바인딩의 경우에는 다른 위치로 적재될 수 있다.

- 스왑핑을 하기 위해서는 보조기억장치가 필요하며, 보통 빠른 디스크를 활용한다.

- 현재는 보통 변형된 형태의 스왑핑만을 사용한다.

9.3 연속적 공간 할당

- 주기억장치는 보통 두 영역으로 분할된다. 한 영역은 운영체제가 사용하며, 다른 영역은 사용자 프로세스가 사용한다. 보통 운영체제는 하위 주소에 위치한다.

- 연속적 공간 할당에서 각 사용자 프로세스는 연속된 단일 영역을 할당받는다.

9.3.1 주기억장치 보호

- 운영체제를 사용자 프로세스로부터 보호해야 하며, 사용자 프로세스를 다른 사용자 프로세스로부터 보호해야 한다.

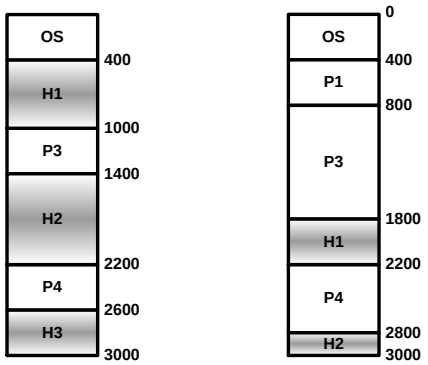
- 이를 위해 기저 레지스터와 한계 레지스터(limit register)를 사용한다.

- 기저 레지스터의 값이 b이고 한계 레지스터의 값이 l이면 논리적 주소 a의 가능한 범위는 $b \leq a \leq b + l$이다. CPU에 의해 생성된 모든 논리적 주소는 사용되기 전에 범위 내에 있는지 검사한다.

9.3.2 주기억장치 할당

- MFT (Multiprogramming with Fixed number of Tasks) 방법
  - 주기억장치를 고정된 크기의 영역으로 분할하여 각 영역을 하나의 프로세스에게 할당한다.
  - 이 경우 다중 프로그래밍의 정도는 분할 영역의 수에 의해 결정된다.
  - 운영체제는 어떤 영역이 사용가능한지를 유지하기 위해 테이블을 사용한다.
  - 이 방법을 MFT라 하며 IBM에서 초창기에 사용한 방식이다. 그러나 현재 이 방법은 사용되지 않고 있다.

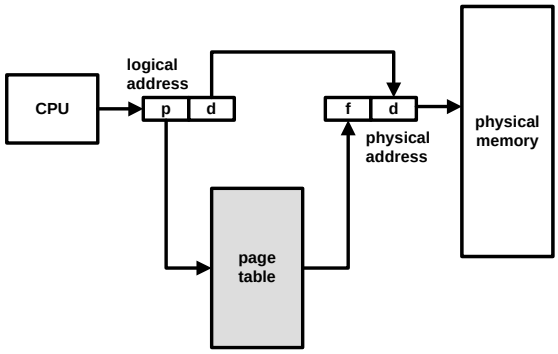
- MVT (Multiprogramming with Variable number of Tasks) 방법
  - 운영체제는 현재 사용되고 있는 공간과 사용되지 않는 공간을 테이블에 유지한다.
  - 처음에는 운영체제가 적재되어 있는 공간을 제외하고는 모두 사용이 가능하며, 사용가능한 영역을 홀(hole)이라 한다.
  - 프로세스가 도착하면 그 프로세스를 수용할 수 있는 크기의 홀을 찾아 할당한다.
  - 프로세스가 실행되기 위해 도착하면 입력 큐(준비완료 큐)에 할당된다.
  - 스케줄링 알고리즘에 따라 프로세스에게 주기억장치를 할당하며, 더 이상 할당할 공간이 없으면 프로세스는 대기하게 된다.
  - 어떤 주어진 시간에 주기억장치를 관찰하면 홀들은 주기억장치 전반에 걸쳐 흩어져 있다.



(a) 동적 기억장치 할당 문제 (b) 외부 단편화

<그림 9.1> MVT 알고리즘

- 동적 기억장치 할당 문제: 여러 홀이 있을 때 어떤 홀에 프로세스를 할당할 것인지를 결정하는 문제
  - **최초 적합(first-fit)**: 충분히 큰 첫 번째로 발견한 홀에 할당한다. 검색은 처음부터 시작할 수 있고, 이전 검색 종료 위치부터 시작할 수 있다.
  - **최적 적합(best-fit)**: 프로세스를 수용할 수 있는 가장 작은 홀에 할당한다. 홀이 정렬되어 있지 않으면 모든 홀을 검사해야 한다.
  - **최악 적합(worst-fit)** 가장 큰 홀에 할당한다. 이 방법을 사용하면 남는 홀의 크기가 가장 크다. 따라서 최적 적합보다는 이것을 다시 활용할 확률이 높다.
- 예) 그림 9.1의 (a)에서 크기가 300 KB인 프로세스가 준비되면 최초 적합인 경우에는 H1에 할당되고, 최적 적합인 경우에는 H3에 할당되며, 최악 적합인 경우에는 H2에 할당된다.
- 시뮬레이션 결과 최초 적합과 최적 적합이 공간 사용 효율이나 할당 시간 측면에서 최악 적합보다는 우수하며, 최초 적합은 최적 적합보다 할당 시간 측면에서 우수하다.
- 위 세 알고리즘의 문제점: 외부 단편화(external fragmentation)가 발생할 수 있다.
- 외부 단편화: 모든 홀을 합하면 프로세스를 수용할 수 있으나 연속 공간이 아니기 때문에 프로세스를 할당할 수 없는 경우를 말한다.
- 예) 그림 9.1의 (b)에서 크기가 500 KB인 프로세스가 준비되면 H1과 H2를 합쳐 600 KB가 빈 공간이지만 연속 공간이 아니기 때문에 할당될 수 없다. 즉, 총 600 KB의 외부 단편화가 있다.
- 외부 단편화를 해결하는 방법에는 compaction이라는 방법이 있다. 이것은 할당되어 있는 프로세스의 적재 위치를 재배치하여 하나의 큰 홀을 얻는 방법이다.



<그림 9.2> 페이징 하드웨어

- MVT에서 어떤 프로세스에게 기억장치 공간을 할당한 후에 아주 작은 홀이 남으면 이것은 현재 할당받는 프로세스에게 할당된 것으로 간주하고 홀로 유지하지 않는다. 이것은 홀에 관한 정보를 유지하는 비용을 줄이기 위함이다. 이와 같은 낭비를 내부 단편화(internal fragmentation)라 한다. 프로세스에게 할당되었지만 사용되지 않는 공간을 말한다.
- 보통 할당된 영역이 N개이면 $0.5N$개의 영역이 단편화 때문에 낭비된다.

9.4 페이징

- 페이징 기법에서는 프로세스에게 연속된 공간을 할당해주지 않아도 된다.
- 현재 가장 널리 사용되고 있는 기법이다.
- 초창기 페이징 기법은 전적으로 하드웨어로 구현하였다. 그러나 최근에는 운영체제와 하드웨어를 밀접하게 연관시켜 구현한다.

9.4.1 기본 개념

- 물리적 기억장치는 고정된 크기의 **프레임(frame)**으로 나눈다.
- 논리적 주소공간도 프레임과 같은 크기의 **페이지(page)**로 나눈다.
- 프로세스가 실행될 때 그것의 페이지는 사용 가능한 어떤 프레임에도 할당될 수 있다.
- 하드웨어 지원: 그림 9.2 참조.
  - CPU가 생성하는 주소: 페이지 번호(p)와 페이지 오프셋(d), 두 필드로 구성된다.
  - 페이지 번호는 페이지 테이블을 참조하는 색인으로 활용되며, 이를 통해 이 페이지가 할당되어 있는 프레임을 알 수 있다.
- 예) 페이지의 크기는 4 바이트이고, 주기억장치의 크기는 32 바이트이다.
  - 주기억장치의 프레임 수: 32/4 = 8(0번부터 7번까지)

  - 프로세스의 페이지 테이블이 다음과 같다고 가정하면

페이지 번호	프레임 번호
0	1
1	4
2	3
3	7

    이 프로세스의 페이지 0은 프레임 1에 할당되어 있고, 페이지 1은 프레임 4에 할당되어 있다.

- 페이징을 사용하면 외부 단편화는 발생하지 않는다. 하지만 내부 단편화는 발생할 수 있다. 프로세스의 마지막 페이지는 프레임 크기보다 작을 수 있다. 최악의 경우 한 프레임의 대부분이 낭비될 수 있다.

- 일반적으로 프로세스마다 프레임의 반 크기 정도의 내부 단편화가 발생한다.

- 내부 단편화를 줄이기 위해 페이지의 크기를 줄일 수 있지만 이렇게 하면 유지해야 하는 페이지 테이블이 커진다.

- 오늘날은 4 KB 또는 8 KB의 페이지 크기를 사용한다.

- 어떤 운영체제는 다중 페이지 크기를 제공한다.

- 페이지 테이블의 한 항목은 보통 4 바이트이다. 이 경우 총 2^{32} 프레임을 지원할 수 있다. 이 때 프레임의 크기가 4 KB이면 총 $2^{44} = 16$ TB의 물리적 주소공간을 사용할 수 있다.

- 페이징에서 주기억장치에 대한 사용자 관점과 실제 물리적 주기억장치가 다르다. 사용자 프로그램은 주기억장치를 하나의 프로그램만 적재되어 있는 하나의 연속적 공간으로 간주한다. 그러나 실제로 사용자 프로그램은 페이지 단위로 나뉘어 흩어져 있다. 이 차이는 주소 번역 하드웨어가 극복해주며, 사용자는 이 사실을 알 필요가 없다.

- 운영체제는 주기억장치를 관리하므로 할당의 자세한 내용을 알고 있어야 한다. 이를 위해 프레임 테이블을 유지한다.

- 프레임 테이블에는 실제 프레임마다 하나의 항이 존재하며, 프레임의 할당여부, 할당되어 있을 경우에는 어떤 프로세스의 몇 번째 페이지가 할당되어 있는지를 유지한다.

- 운영체제는 또한 각 프로세스마다 페이지 테이블을 유지한다. 이 페이지 테이블은 보통 PCB 내에 유지되며, 문맥전환을 할 때 필요하다. 따라서 페이징은 문맥전환에 소요되는 시간을 증가시킨다.

9.4.2 하드웨어 지원

- 페이지 테이블의 하드웨어 구현 방법
  - 전용 레지스터 집합을 사용하는 방법: 페이지 테이블이 작은 경우에만 사용할 수 있음
  - 주기억장치의 특정 영역에 페이지 테이블을 저장하고, 이것을 가리키는 주소만 레지스터에 저장한다. 이 레지스터를 PTBR(Page-Table Base Register)이라 한다.
    - 문제점: 주기억장치를 접근하기 위해 항상 두 번의 접근이 필요하므로 접근속도가 느리다.
  - 전용 캐시를 사용하는 방법
    - 이 캐시를 **Translation Look-aside Buffer** (**TLB**)라 하며, 보통 캐시와 마찬가지로 연관 매핑(associative mapping) 방식을 사용한다.
    - TLB는 크기가 제한적이며, 비용이 비싸다.
    - 따라서 TLB에는 페이지 테이블에 일부만 가지고 있으며, 캐시처럼 먼저 TLB를 검색한 다음에 찾고자 하는 페이지 정보가 없으면 주기억장치에 있는 페이지 테이블을 참조한다.
    - TLB에 있는 어떤 정보들은 제거될 수 없도록 한다. 이것을 wired down 되었다고 하며, 보통 커널과 관련된 페이지 정보는 TLB에서 제거되지 않는다.
    - 어떤 TLB 구현들은 ASID(Address-Space IDentifier)를 페이지 정보와 함께 저장할 수 있도록 해준다. ASID는 프로세스 간의 주기억장치 보호 역할을 해주며, 여러 프로세스의 페이지 정보가 TLB에 함께 저장될 수 있도록 해준다.

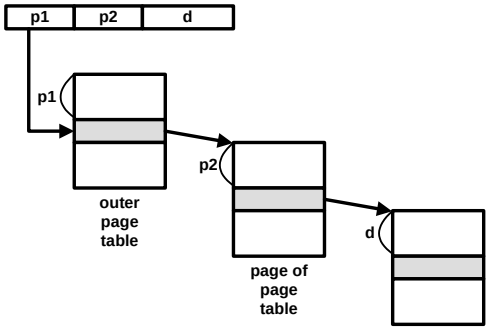
9.4.3 보호

- 페이징 기법을 사용하는 환경에서 주기억장치 보호는 보호 비트를 이용한다.

- 보호 비트는 페이지 테이블에 유지된다.

- 각 페이지마다 한 비트를 이용하여 페이지가 읽기-쓰기 또는 읽기-전용인지를 나타낼 수 있다.
  - 모든 주기억장치 참조는 페이지 테이블을 참조하므로 이 때 보호비트를 이용하여 읽기-전용 페이지에 대한 쓰기 시도를 막을 수 있다.

- 일반적으로 페이지 테이블의 각 항마다 유효비트가 붙어있다. 이 비트가 설정되어 있으면 이 항에 해당하는 페이지는 프로세스의 논리 주소 공간에 포함됨을 나타낸다.
  - 유효비트 대신에 PTLR(Page Table Length Register)를 사용하여 페이지 테이블의 크기를 나타내는 경우도 있다.



<그림 9.3> 두 단계 페이징 구조에서 주소 해석

9.4.4 페이지 테이블의 구조

9.4.4.1 계층구조 페이징

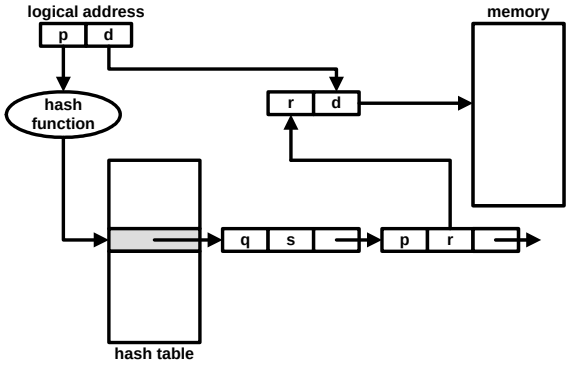
- 현대 컴퓨터 시스템은 매우 큰 논리 주소 공간(2^{32}에서 2^{64})을 제공한다. 한 페이지의 크기가 4 KB(2^{12})이면 32 비트 컴퓨터의 경우 페이지 테이블은 2^{20} 항이 필요하며, 각 항이 4 바이트이면 4 MB가 필요하다. 따라서 페이지 테이블을 연속적 주소 공간에 저장하는 것은 힘들다.
- 계층구조 페이징이란 페이지 테이블도 페이징을 하는 페이징 기법이다.
- 이 기법에서 논리 주소는 다음과 같이 구성된다.

page number		page offset
p1	p2	d
10	10	12

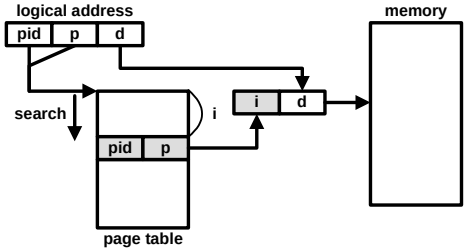
  여기서 p1은 첫 레벨 페이지 테이블에 대한 색인이며, p2는 두 번째 레벨 페이지 테이블 대한 색인이고, d는 실제 페이지 내의 오프셋이다. 이때 주소 참조는 그림 9.3과 같이 이루어진다. 이런 방식을 forward-mapped 페이지 테이블이라 한다.
- 64 비트 논리 주소 공간을 제공하는 시스템에서는 두 단계 페이징 방식으로는 부족하다.
  - 두 단계 페이징 방식을 이용하면 p1은 42 비트가 필요하다. 따라서 첫 레벨 페이지 테이블은 2^{42}개의 항이 필요하다. 즉, 4 TB가 필요하다.
  - 그러므로 다 단계 페이징 방식을 사용해야 하지만 단계가 깊을 수록 주소를 참조하는데 많은 시간이 소요된다.

9.4.4.2 해시된 페이지 테이블

- **해시 테이블(hash table)**의 각 항은 같은 위치로 해시되는 페이지 번호의 연결 리스트로 구성된다. 연결 리스트의 각 요소는 다음과 같은 세 가지 필드로 구성된다.
  - 가상 페이지 번호
  - 페이지 프레임의 값
  - 다음 요소를 가리키는 포인터



<그림 9.4> 해시된 페이지 테이블

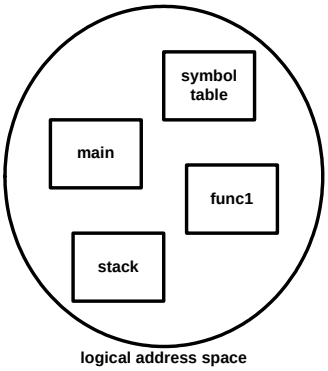


<그림 9.5> 역 페이지 테이블

- 주소 해석 방법
  - 주소에 있는 가상 페이지 번호를 해시하여 해당하는 해시 테이블의 위치를 계산한다.
  - 그 위치에 있는 연결 리스트를 검색하여 페이지 프레임 값을 얻는다.
- 단계가 깊은 다 단계 페이징 방식보다는 주소를 참조하는데 소요되는 시간이 평균적으로 적다.

9.4.4.3 역 페이지 테이블

- 지금까지의 페이징 기법은 각 프로세스마다 별도의 페이지 테이블을 사용하였다. 이 페이지 테이블은 프로세스가 사용하는 각 페이지에 대한 항(1 MB의 프로세스이고, 4 KB의 페이지를 사용하면 2^8개)으로 구성되어 있거나 페이지 사용 여부와 상관없이 전체 주소 공간에 대한 항(32 비트 주소이면 2^{32}개)으로 구성되어 있다. 그러나 이 방법은 매우 큰 공간을 필요로 한다.
- 이것에 대한 해결책으로 역 페이지 테이블 기법을 사용할 수 있다.
- 역 페이지 테이블은 주기억장치의 각 프레임에 대한 항만을 가지고 있다. 이 항에는 이 프레임에 저장되어 있는 가상 주소와 이것을 사용하는 프로세스의 식별자로 구성되어 있다.
- 주소 해석 방법
  - 각 가상 주소는 세 개의 필드(프로세스 식별자, 페이지 번호, 오프셋)로 구성되어 있다.



<그림 9.6> 사용자 관점의 프로그램

- 프로세스 식별자와 페이지 번호를 이용하여 페이지 테이블을 검색한다. 일치하는 것을 발견하면 그 항의 위치가 프레임의 위치가 된다.
- 문제점: 페이지 테이블을 검색하는 비용이 많다. 이것을 극복하기 위해 해시 테이블을 활용할 수 있다.

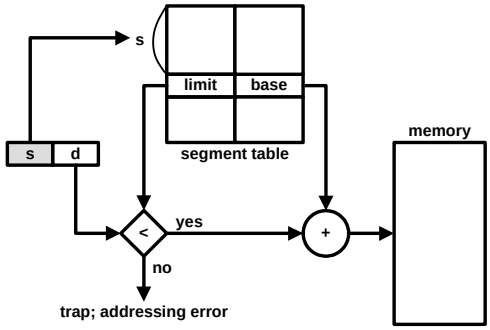
9.4.5 공유 페이지

- 페이징 기법의 또 다른 장점은 공통 코드의 공유 가능성이다.
- 예) 시스템에 40명의 사용자가 사용 중이고, 이들 모두가 150 KB의 코드와 50 KB의 데이터 공간이 필요한 문서 편집기를 사용하고 있다. 그러면 8,000 KB 공간이 필요하다. 그러나 만약 이 코드가 재진입 가능 코드이면 150 KB의 코드 부분은 모든 사용자가 공유할 수 있다. 따라서 총 2150 KB가 필요하다.
  - 재진입 코드는 자체 수정이 가능하지 않아야 한다. 즉, 스스로 수행되는 동안에 변경되지 않아야 한다.
  - 코드를 공유하기 위해서는 반드시 재진입이 가능해야 한다.
- 역 페이지 파일을 사용하면 코드를 공유하기가 어렵다. 역 페이지 파일 방식에서는 개별 프로세스의 페이지 테이블을 유지하지 않으므로 모든 프로세스가 하나의 큰 단일 논리 주소 공간을 사용하게 된다. 따라서 두 프로세스가 페이지를 공유하기 위해서는 그 페이지에 대해 같은 가상 주소를 사용해야 하므로 공유가 쉽지 않다.

9.5 세그먼테이션

9.5.1 기본 방법

- 보통 사용자들은 주기억장치를 연속적 선형 공간으로 생각하지 않고, 보통 주기억장치를 순서가 없는 다양한 크기의 세그먼트의 집합으로 생각한다. 그림 9.6 참조.



<그림 9.7> 세그먼테이션 하드웨어

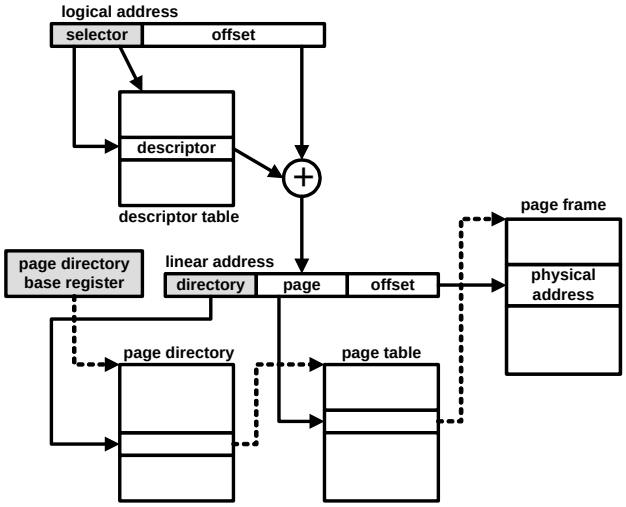
- 세그먼테이션은 이런 사용자 관점을 지원해주는 주기억장치 관리 기법이다.
- 세그먼테이션에서 논리 주소 공간은 세그먼트의 집합으로 구성되며, 각 세그먼트는 이름과 길이를 가진다.
- 사용자는 세그먼트 이름과 오프셋을 이용하여 주소를 지정한다.
- 컴파일러가 사용자 프로그램을 번역할 때 자동적으로 세그먼트를 구성해준다.

9.5.2 하드웨어

- 세그먼트 주소와 물리적 주소 간에 매핑
  - 세그먼트 테이블을 사용한다.
  - 이 테이블의 각 항은 세그먼트의 기저와 한계로 구성되어 있다. 세그먼트의 기저에는 세그먼트의 물리적 시작 주소가 기록되어 있으며, 한계에는 세그먼크의 길이가 기록되어 있다.

9.5.3 보호와 공유

- 세그먼트는 논리적 구성 단위인 동시에 물리적 구성 단위이다. 따라서 일반적으로 세그먼트는 프로그램의 의미적으로 정의된 일부분이다. 따라서 보호 용도를 정의하기가 쉽다.
  - 세그먼트는 크게 데이터 부분과 명령어 부분으로 구분될 수 있으며, 명령어 부분은 읽기-전용 또는 실행-전용으로 지정하여 보호할 수 있다.
  - 페이징 기법에서 언급된 것과 같은 보호비트를 각 세그먼트와 연관하여 세그먼트를 적절하게 보호할 수 있다.
  - 배열을 하나의 세그먼트로 관리하면 자동으로 배열 경계를 검사할 수 있다.
- 세그먼트는 페이지와 마찬가지로 코드와 데이터의 공유가 가능하다. 또한 이 공유는 세그먼트 단위로 이루어질 수 있다.



<그림 9.8> 80386 주소 해석

  - 문제점: 일반적으로 코드 세그먼트는 자체 참조(예, 조건부 분기)를 포함하고 있다. 이 주소는 세그먼트와 오프셋으로 구성되므로 모든 프로세스는 공유하는 세그먼트에 대해서는 같은 이름(번호)를 사용해야 한다.
  - 해결책: 세그먼트 내에서는 간접 주소(현재 주소에서 오프셋) 방식을 사용한다.

9.5.4 단편화

- 페이징 기법은 고정된 크기의 페이지를 사용하므로 외부 단편화가 발생하지 않지만 세그먼트는 가변 길이이므로 외부 단편화가 발생할 수 있다.

9.6 페이징과 세그먼테이션의 결합

9.6.1 인텔 80386

- 각 프로세스는 최대 16 K개의 세그먼트를 가질 수 있다.
- 각 세그먼트는 최대 4 GB가 될 수 있다. 이 세그먼트는 페이징된다.
- 페이지의 크기는 4 KB이다.
- 한 프로세스의 논리 주소 공간은 크게 두 영역으로 분할된다.
  - 영역 1. 프로세스가 독점적으로 사용하는 세그먼트들: 최대 8 K개의 세그먼트
  - 영역 2. 다른 프로세스와 공유하는 세그먼트들: 최대 8 K개의 세그먼트
- 영역 1에 관한 정보는 LDT (Local Descriptor Table)에 유지되며, 영역 2에 관한 정보는 GDT (Global Descriptor Table)에 유지된다. 이 데이블의 각 항은 8 바이트이며, 특정 세그먼트의 기저와 한계에 관한 정보가 기록된다.
- 논리 주소는 선택자(16 비트)와 오프셋(32 비트) 쌍으로 구성되어 있으며, 선택자는 다음과 같은 세 개의 필드로 구성되어 있다.
  - 세그먼트 번호 s(13 비트)
  - 구분자 g(1 비트): LDT와 GDT를 구분
  - 보호 관련 비트 p(2 비트)
- 80386은 6개의 세그먼트 레지스터를 제공한다. 따라서 한 번에 6개의 세그먼트를 참조할 수 있다.
- 80386은 6개의 8 바이트 크기의 레지스터로 구성된 캐시를 사용한다. 이 캐시는 LDT 또는 GDT의 항을 캐시한다.
- 80386은 두 단계 페이징 방식을 사용한다.