**8. main memory**

# 8.1.1 기본 하드웨어

- 주 메모리와 레지스터들은 CPU가 직접 접근할 수 있는 유일한 저장장치

- 레지스터 접근은 CPU 한 클럭 이하 안에 일어난다.

- 주 메모리에 접근은 지연(stall)을 발생 시키며 이에 대한 해결책으로 캐시가 있다.

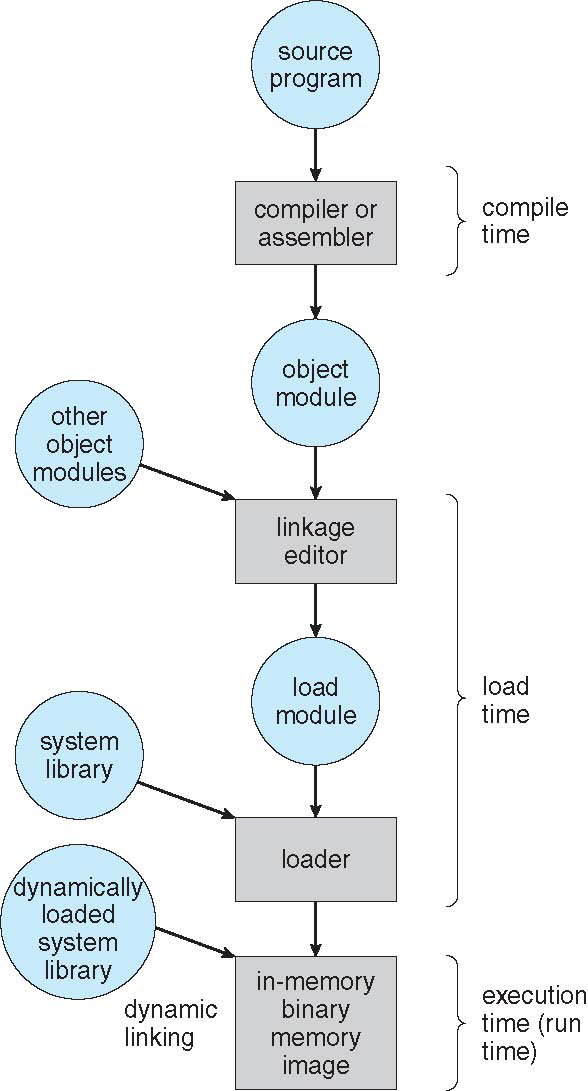
- 베이스 레지스터 = 300040, 상한(limit) 레지스터 = 120900이라면 이 프로그램은 300040 ~ 420940까지의 주소를 접근할 수 있다. (베이스, 상한 레지스터는 특권명령을 통한 커널모드에서만 값을 변경할 수 있다)

- 이러한 메모리 공간의 보호를 통해 다른 사용자 프로그램의 코드나 자료구조의 수정을 막음(trap 발생)

# 8.1.2 주소의 할당

- 프로그램이 실행되기 위해서는 주 메모리로 올라와서 ‘프로세스’가 되어야함.

- 프로세스들은 입력 큐를 이루어 실행되며 프로세스가 종료되면 사용된 기억 공간이 가용 공간이 되어 다른 프로세스를 위해 사용됨.

- 논리주소(심볼) 100 ~ 500 🡪 절대주소 100100 ~ 100500 이런 식으로 바인딩 되며 실행 시점에 따라 구분됨

1. 컴파일 시간(compile time) 바인딩 :

메모리에 들어갈 위치를 컴파일 시간에 미리 알 수 있다면 컴파일러는 절대코드를 생성하여 번역할 코드를 그 위치부터 시작. 다만 이 위치가 변경되어야 한다면 다시 컴파일 해야 함

2. 적재 시간(load time) 바인딩 :

컴파일러는 이진 코드는 재배치 가능한 코드로 만들어야 하며 프로그램이 주 메모리에 적재될 때 심볼이 진짜 주소로 바인딩 됨.

3. 실행 시간(Execution time) 바인딩 :

프로세스가 실행 중에 메모리의 다른 세그먼트로 옮겨질 수 있다면 바인딩이 실행 시간까지 (지연/허용)됨 [ 베이스, 상한 레지스터 등 특별한 하드웨어가 필요]

# 8.1.3 논리 vs 물리 주소 공간

* 논리주소(Logical address) == 가상주소(virtual address) 같은 말
* 여기서는 베이스 레지스터를 재배치(relocation) 레지스터라고 부름
* 논리주소 공간과 물리주소 공간이 분리되어 메모리 관리
* 논리주소 🡪 물리주소 변환은 MMU(Memory-Management Unit)이 해줌
* 물리주소 기준 값 = R, 논리주소 최댓값 = max 일 때 물리주소 공간은 R ~ R+max
* 사용자 프로그램은 실제 물리 주소를 알 수 없다.

# 8.1.4 동적 적재 (Dynamic loading)

* 프로그램을 통째로 적재 시키는 것보다 메모리 공간의 효율적인 이용을 위해 사용하는 부분만 적재하는 방법

1. 각 루틴은 호출 전까지 재배치 가능한 상태로 디스크에서 대기.
2. 호출된 루틴이 메모리에 이미 적재되어 있지 않다면 재배치가능연결적재기(relocatable linking loader)가 불려져 루틴을 메모리로 가져오고 테이블에 기록한다.
3. 그 후 CPU제어는 중단되었던 루틴으로 보내진다.

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

* 동적 적재에서는 적재(loading)가 실행 시까지 미뤄졌다면 동적 연결은 연결(linking)이 실행 시까지 미뤄지는 것.
* 동적 적재와 달리 동적 연결은 운영체제의 도움이 필요함.
* 정적 연결로는 하나의 라이브러리를 하나의 이진프로그램 이미지가 하나씩 가지게 되는데 (ex. printf()) 같은 라이브러리를 10개의 프로세스가 사용한다면 printf() 라이브러리 코드가 10개나 필요함 🡪 동적 연결을 한다면 라이브러리를 부르는 곳마다 스텁(stub. 라이브러리를 어떻게 찾을 것인가 알려주는 코드조각)이 생겨 라이브러리가 메모리에 있는 지 확인하고 없으면 디스크에서 가져온다. 가져오면 자신을 그 루틴의 번지로 대체하여 이후로 호출될 때는 루틴을 수행하면 됨. (ex. printf()) 같은 라이브러리를 10개의 프로세스가 사용한다면 printf() 라이브러리 코드가 1개면 끝.
* 라이브러리가 새로운 버전으로 바뀔 때 유용함. 동적 연결이 없다면 라이브러리를 사용하는 모든 프로그램을 새로 연결 시켜야 함. [공유 라이브러리(Shared Library)

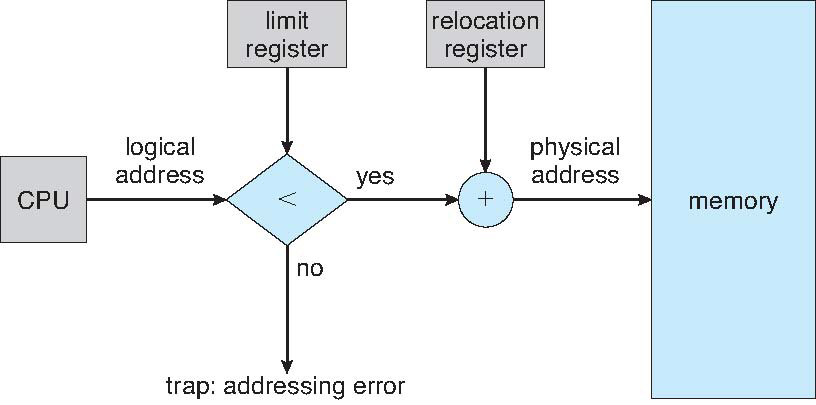
# 8.2 스와핑(Swapping)

* 주 메모리가 가득 찼을 때 프로그램을 디스크로 쫓아내고 여유가 생겼을 때 다시 불러오는 기능
* 라운드 로비 스케줄링의 경우에 CPU할당 시간이 만료된 프로세스를 스왑.
* [롤인(roll-in), 롤아웃(roll-out)]낮은 우선순위의 프로세스를 스왑하고 높은 우선순위의 프로세스가 종료되면 다시 불러들임.
* ㄱ) 바인딩이 적재 시간에 이루어지는 프로세스는 스왑되어 돌아올 때 이전과 같은 주소로 돌아와야 함.
* ㄴ) 바인딩이 실행 시간에 이루어진다면 어디로든 돌아갈 수 있음
* 디스패처(dispatcher)가 준비완료큐(ready queue)에서 다음 프로세스를 메모리에 올리고 메모리에 공간이 부족하다면 메모리에 올라와 있는 프로세스를 스왑시킴. 이후 제어권을 프로세스에게 넘긴다.
* 스와핑은 문맥교환시간(context-swith time)이 오래 걸림.
* 🡪 프로세스의 크기 = 10MB, 보조메모리 전송률 = 40MB, 회전 지연시간 = 8ms라면, 스왑인 10/40초(250ms), 스왑아웃 10/40초가 되어 516ms가 걸림.
* 🡪효율을 위해 각 프로세스의 실행 시간은 516ms보다 커야함.
* 스왑 시간을 줄이기 위해 운영체제에게 메모리의 요구 사항의 변화를 알려줄 수 있는 request\_memory(), release\_memory()같은 시스템 호출이 필요함.
* 입출력이 진행되는 동안 스왑하면 안된다.
* 해결1) 입출력이 진행되는 동안 스왑하지 않는다.
* 해결2) 입출력을 직접 하지 않고 운영체제의 버퍼만 하도록 한다.
* 대부분의 모바일 운영체제는 지원하지 않는 기능, (코드, 데이터, … 중에서 코드는 변하지 않으므로 코드 부분의 메모리를 뺐음)

# 8.3 연속 메모리 할당 (Contigous Memory Allocation)

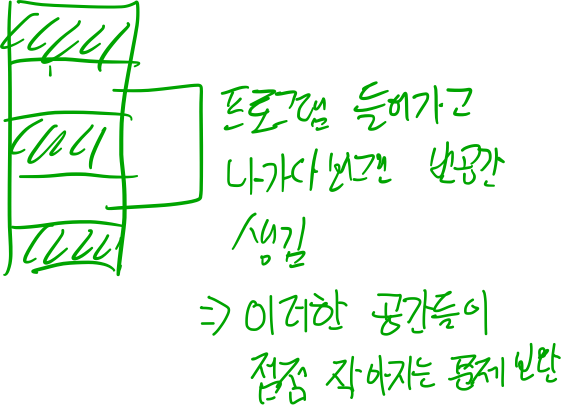
: 흔히 인터럽트 벡터가 0번지에 위치하기 때문에 운영체제는 하위 메모리에 위치시킨다.

# 8.3.1 메모리 사상(mapping)과 보호(protection)

* MMU가 동적으로 논리 주소에 재배치 레지스터의 값을 더함으로써 주호를 변환하는 역할을 한다.
* CPU 스케줄러가 다음으로 수행할 프로세스를 선택할 때 디스패처는 문맥 교환의 일환으로 재배치 레지스터와 상한 레지스터에 정확한 값을 적재한다. 이로써 다른 사용자 프로그램의 접근으로부터 보호할 수 있다.
* 재배치 레지스터를 사용함으로써 운영체제의 크기는 실행중이더라도 바뀔 수 있다. 이 기능은 운영체제에서 더이상 필요하지 않은 일시적 운영 체제 코드(transient OS code)를 메모리에서 지워 운영체제의 크기를 줄일 수 있다.

# 8.3.2 메모리 할당(Memory Allocation)

1. 메모리를 똑같은 크기로 분할하여 하나당 하나의 프로세스에 할당, 사용이 끝난 분할은 다른 프로세스에게 할당 가능, 이 때 분할의 개수를 다중 프로그래밍 정도(multiprogramming degree)라고 부름
2. [변수 분할]기법에서는 운영체제가 메모리의 사용되고 있는 부분을 파악하는 테이블을 유지한다.

* 초기에 메모리 공간은 한 개의 큰 블록으로 간주되며 한 개의 공간(hole)이 있다고 표현한다.
* 운영체제는 입력 큐에서 각 프로세스를 꺼내오며 빈 공간보다 많은 메모리를 요구하면 미룰 수 있고 더 적은 메모리를 요구하면 빈 공간으로 쪼개 할당할 수 있다.
* 이러한 기법은 동적 공간 할당 문제(dynamic storage allocation problem)의 특별한 한 예다. 일련의 공간으로부터 n바이트 블록 요구를 어떻게 만족시킬지에 대한 해결책의 일반적인 기법들이 있다
* 최초 적합(First-fit) : 앞이나 뒤에서부터 검색을 하여 첫 번째로 사용 가능한 공간을 할당
* 최적 적합(Best-fit) : 사용 가능한 공간 중 가장 작은 것을 할당
* 최악 적합(Worst-fit) : 가장 큰 공간을 할당



* 시뮬레이션 결과 최초 > 최적 > 최악 순으로 좋음

# 8.3.3 단편화(Fragmentation)

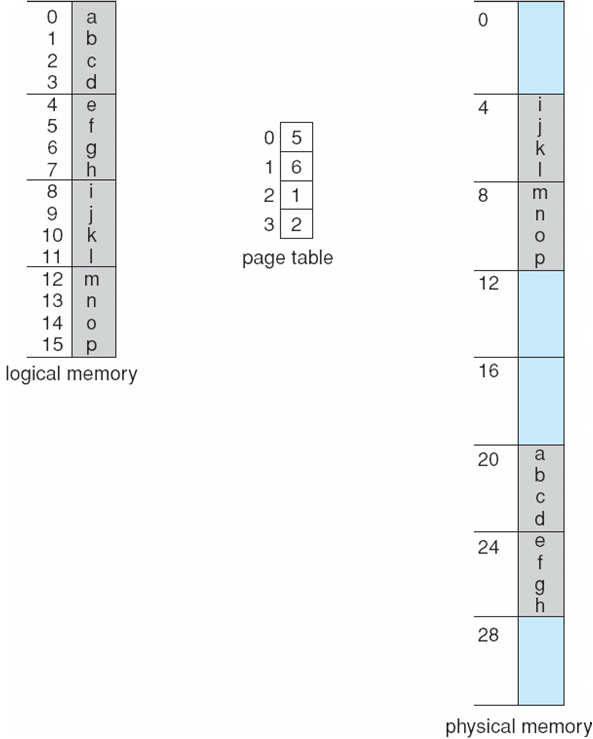
: fagment는 공간 중 사용 못한게 되는 부분을 뜻함

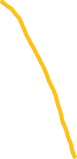
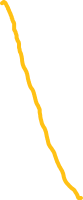
* 외부 단편화(External fragmentation) : 프로세스들이 메모리에 적재되고 제거되다보면 빈 공간들이 작은 조각으로 여기저기 분산될 수도 있음.
* 이에 대한 고려 요소로 위 또는 아래부터 할당할 것인지 고려해야 한다.
* 최초적합의 분석 결과 N개의 블록이 할당되었을 때 0.5N개의 블록이 단편화로 손실됨, 즉 메모리의 1/3이 쓸 수 없다. [50% 규칙]
* 내부 단편화(internal fragmentation) : 한 블록의 크기가 100인데 프로세스가 98을 요구한다면 2만큼 낭비됨. 이러한 경우 2바이트를 관리하는 것이 낭비하는 것보다 비용이 큼.
* 해결법으로 압축(compaction)이 있다. 메모리의 내용들은 위 또는 아래로 몰아서 자유공간을 하나의 큰 블록으로 만드는 것.
* 프로세스 내의 모든 주소들이 동적으로 재배치 되어야한다. 따라서 재배치가 동적으로 이루어지는 경우만 가능
* 단순히 모든 프로세스를 메모리의 한쪽 끝으로 이동시키는 알고리즘은 비용이 매우 큼

# 8.4 페이징(Paging)

: 논리 주소 공간이 한 연속적인 공간에 모여 있어야 하는 제약을 없앨 수 있다.

# 8.4.1 기본 방법

* 물리 메모리는 프레임(frame), 논리 메모리는 페이지(page)라 불리는 같은 크기의 블록으로 나누어진다.
* 논리주소는 [페이지번호 – 변위(offset)]로 나타난다.
* 페이지의 크기는 하드웨어에 의해 정의되며, 논리주소 공간의 크기가 2n, 페이지의 크키가 2m 이라면 상위 n-m비트는 페이지 번호를 나타내며 하위 n비트는 변위를 나타낸다.
* 페이징은 외부 단편화는 발생시키지 않지만 거의 모든 경우 내부 단편화가 발생한다.
* (ex. 프로세스가 72,766을 요구하고 페이지의 크기가 2,048이라면 72,766 – (2,048 \* 35) = 962의 내부 단편화가 발생)
* 내부 단편화를 줄이기 위해 페이지의 크기를 줄이는 것이 좋지만, 그에 따라 페이지 테이블의 크기가 커지는 문제 발생
* 페이지 테이블 내의 각 항목이 4 바이트이면 232개의 물리 페이지 프레임을 가리킬 수 있다. 프레임 크기가 4KB이면 244바이트개의 물리 주소를 저장할 수 있다.
* 사용자는 메모리가 연속된 공간이라고 인식하지만 실제로 페이징을 하면 메모리의 여러 곳에 프로그램이 프레임 단위로 분산되어 있다.
* 운영체제는 메모리를 관리하기 위해 프레임들에 대한 정보를 프레임 테이블에 보관하며, 프레임 테이블은 각 프레임당 하나의 항목을 가지고 할당 정보를 나타낸다.
* 운영체제는 각 사용자에 대해 페이지 테이블의 복사본을 유지해야 한다
* 페이징은 문맥 교환 시간을 증가시킨다.



# 8.4.2 하드웨어 지원

* 대부분의 OS는 프로세스마다 하나의 페이지 테이블을 할당한다.
* 페이지 테이블을 가리키는 포인터는 다른 레지스터들의 값과 함께 저장되어 디스패처에 의해 다시 적재될 때 이전의 작업을 이어서 할 수 있게 한다.
* 페이지 테이블을 레지스터의 집합으로 구현하는 것이 좋지만 페이지 테이블이 너무 많아 불가능, 따라서 페이지 테이블을 주 메모리에 저장하고 페이지 테이블 기준 레지스터(PTBR: Page-Table Base Register)로 페이지 테이블을 가리키도록 한다. 다른 페이지 테이블을 사용하려면 이 레지스터만 바꾸면 돼서 문맥 교환 시간을 줄일 수 있다.
* 이 방식의 문제점은 메모리 접근 시간. (메모리 접근 + 변환된 실제 주소로 접근 = 2번의 메모리 접근 🡪 두 배로 느려짐)
* 해결법으로 TLB(Translation Look-aside Buffer)라는 소형 캐시를 사용
* TLB는 키와 값으로 구성되어 있다.
* 페이지를 찾아달라는 요청이 들어오면 TLB는 동시에 여러 키들과 비교하여 찾아줌
* 그런데 TLB는 매우 비싸서 64 ~ 1024 항목 수를 가지기 때문에 모든 페이지 테이블을 가지고 있을 수 없다. 🡪 TLB에서 찾지 못한 페이지 테이블은 메모리에서 찾아야함[TLB미스]
* TLB미스로 찾아낸 페이지 테이블을 TLB에 추가하여 다음에 빨리 찾을 수 있도록 함
* TLB가 가득 찼을 때 헌 항목을 빼는 교체 작업이 필요
* 일부 항목들은 고정(wire down) 시킨다. (ex. 커널 코드)
* 어떤 TLB는 ASID(address-space identifier)를 저장하여 TLB항목이 어떤 프로세스에 속한 것인지 알려주며, 프로세스의 정보를 보호하기 위해 사용됨.
* ASID가 일치하지 않을 때 TLB미스로 처리
* ASID가 없을 경우 프로세스가 바뀌면 다른 프로세스를 손상시키지 않도록 TLB를 flush시켜야 함
* TLB에서 페이지 번호가 발견되는 비율을 적중률(hit ratio)라고 부름.
* 메모리접급(100ns), TLB탐색(20ns), 적중률 80%일 때 유효 메모리 접근 시간(effective memory access time)은 {0.8 \* (20 + 100)} + {0.2 \* (20 + 100 + 100) = 140ns이다.
* 적중률은 TLB의 크기와 관계있다.

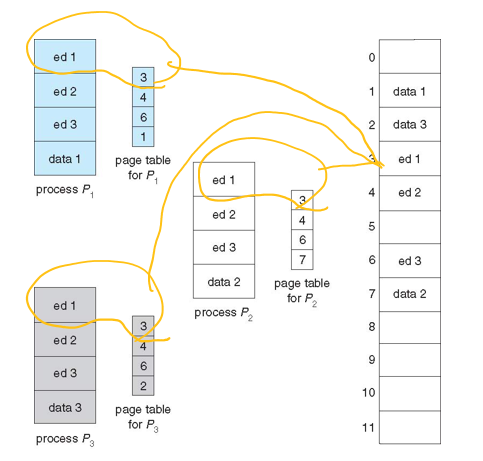
# 8.4.3 보호(Protection)

* 메모리의 보호를 위해 보호비트를 이용하여 읽기쓰기, 읽기 전용을 정의할 수 있다.
* 페이지 테이블의 각 항목에는 유효/무효(valid/invalid)라는 비트가 더 있다
* 14비트(0~16,383) 주소공간을 갖는 시스템에서 10,468의 주소만을 사용할 수 있고 페이지의 크기가 2KB일 때, 페이지 0 ~ 5(0~12,288)의 주소는 유효하지만 6, 7은 무효로 설정되어 트랩(invalid page reference)를 발생시키며, 페이지 5에 대해서는 10,468까지의 주소 접근은 유효하지만 10,469~16,383까지는 무효하다.
* PLTR(page-table length register)를 이용하여 페이지 테이블의 크기를 나타내기도 한다.

# 8.4.4 공유 페이지(Shared page)

* 페이징의 또다른 장점으로 코드를 쉽게 공유할 수 있다는 점이 있다.
* 만약 문서 편집기가 150의 코드와 50의 자료로 구성되어 있다면 40명이 사용하기 위해 총 8,000이 필요한데 재진입 가능 코드(reentrant code 또는 pure code)이라면 문서 편집기 코드를 공유하여 150 + (50 \* 40) = 2,150의 공간으로 구성할 수 있다.
* 이러한 방법은 운영체제에서 코드의 정확성을 보호해주어야 한다.





# 8.5 페이지 테이블의 구조

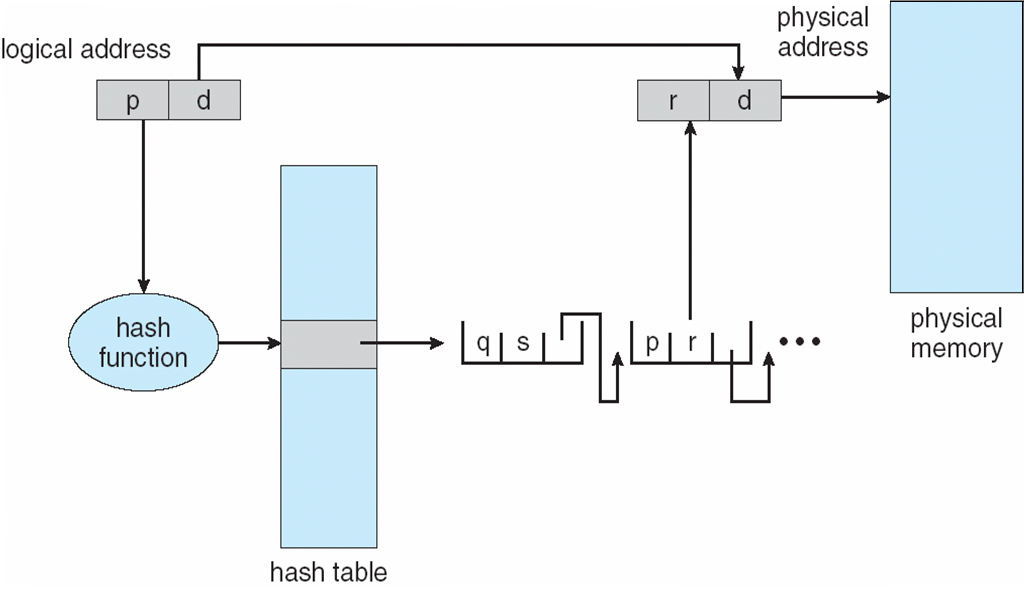
# 8.5.1 계층적 페이징(Hierarchical Paging)

: 요즈음의 컴퓨터는 매우 큰 주소 공간을 가지므로 많은 페이지 테이블이 필요하다. (ex. 32비트 시스템에서 페이지의 크기가 4KB라면 (232/212)개의 페이지가 필요하여 각 프로세스는 페이지 테이블만을 위해서도 4MB의 공간이 필요하다.

* 2단계 페이징 기법(two-level paging scheme) : 페이지를 페이징하는 방법
* VAX 구조가 이러한 방법을 지원한다.

|  |  |  |
| --- | --- | --- |
| 영역 (2bit) | 페이지 ( 21bit) | 변위 (9bit) |

* 4개의 영역으로 페이지를 페이징한 VAX의 구조는 위와 같음.
* 하지만 이러한 방법은 64비트 시스템에서 바깥 페이지가 커지므로 적절하지 않다.
* 3단계 페이징 기법 : 2단계 페이징에서 바깥 페이지를 하나 더 추가한 것
* 너무 많은 메모리 접근을 요구하여 비현실적인 방법임.
* 포워드 사상(forward-mapped) 테이블이라고도 부름

# 8.5.2 해시형 페이지 테이블 (Hashed Page Table)

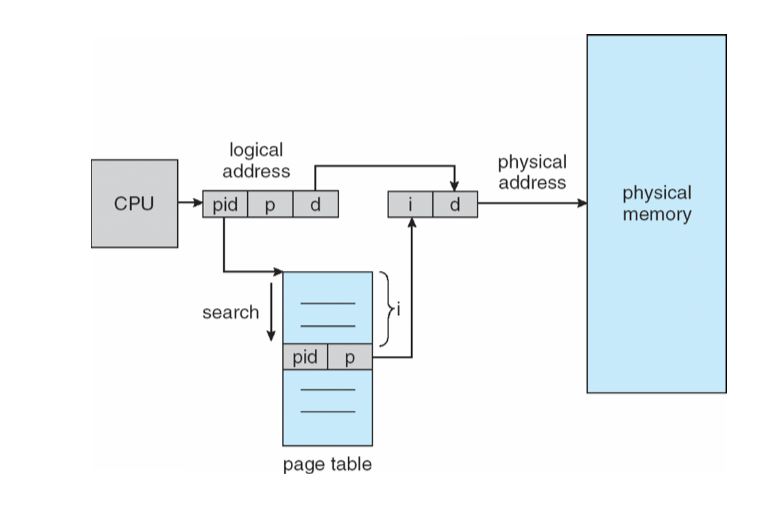
: [가상 페이지 번호 – 프레임 번호 – 연결리스트의 다음 원소 포인터]의 구조를 갖는다.

* 가상 페이지 번호로부터 해싱한 후 연결리스트를 따라 가상페이지와 일치하는 원소를 찾고 일치하면 그에 대응하는 페이지 프레임 번호를 가져와 물리 주소를 얻는다.
* 이 방법을 64비트 시스템을 위해 약간 변형하면 클러스터형 페이지 테이블(Clustered page table)을 만들 수 있다.
* 해시형 테이블의 각 항목들이 한 개의 페이지만 가르키며 각 항목들이 여러 개의 페이지를 가진다.
* Sparse(여기저기 퍼져있는) 주소 공간에 유용하게 사용됨

# 8.5.3 역 페이지 테이블 (Inverted page table)

: 페이지 테이블은 가상 주소에 대해 오름차순으로 정렬되어 있고 이것을 계산을 통해 빠르게 접근할 수 있다. 하지만 테이블 크기가 굉장히 커서 엄청난 메모리 공간을 점유하게 된다.

* 역 페이지 테이블은 메모리 프레임마다 한 항목씩 할당하며 항목은 (프로세스id, 페이지 주소)로 되어있다.
* 이 방법은 물리 프레임에 대응하는 항목만 테이블에 저장하기 때문에 메모리의 공간을 적게 사용한다.
* 다만 탐색을 테이블 전체를 해야 하기 때문에 오래 걸린다.
* 해싱을 통해 해결할 수 있으며 그래도 느리면 TLB를 이용해 최근에 사용된 레지스터부터 검색할 수 있다.
* 메모리의 공유는 하나의 물리영역에 사상되는 여러 개의 가상 주소를 통해 구현되는데 모든 물리 페이지에 대해 하나의 가상 주소를 가지므로 역 페이지에서는 사용할 수 없다.
* 사상되지 못한 다른 가상주소가 공유 영역을 참조하게 되면 page fault 발생.



# 8.6 세그먼테이션 (Segmentation)

: 코드, 전역변수, 스택, 라이브러리, …은 같은 크기를 가지고 있지 않다 따라서 사용자 관점에서 메모리를 바라보기 편한 방법이다.

# 8.6.1 기본 방법

* 유연한 크기를 가진 세그먼트들의 집합으로 논리 주소를 나눈다.
* 논리 주소는 (세그먼트 번호, offset)으로 구성된다.

# 8.6.2 하드웨어

* STBR (Segment-table base register)
* STLR (Segment-table length register)
* 논리 주소가 STBR ~ (STBR + STLR) 사이에 있어야 함. 그렇지 않으면 트랩 발생

