**운영체제 7장부터**

2015253039 권진우

**7장 메모리 관리**

배경

-**프로그램--디스크(SSD,HDD)** 🡨🡪 **메모리(RAM)--프로세스**

- *프로그램은 디스크*에서 🡪 메모리로 적재되어 프로세스로 실행됨

- **PCB** : Process Control Block : OS가 관리하는 프로세스 **자료구조/** 프로세스 정보 저장

**CPU가 직접 접근** 가능한 기억장치 : CPU register, CPU Cache, Main Memory(RAM)

- CPU register : **1Clock**에 접근가능

- Main memory : **여러 Clock(**CAS지연+메모리 클럭에 따라**)만에 접근** -> Process stall(**프로세스 멈춤)발생**

-> CPU와 메모리의 병목현상(속도차이로 인한)을 줄이기 위해 **Cache 메모리**가 사이에 존재

-> **Process stall 빈도 감소**

**메모리 보호** : 메모리에서 특정한 영역을 보호 🡨 **올바른 동작과 보호를 위해**

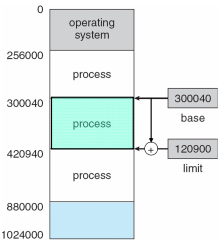
<- 하드웨어 지원(소프트웨어 코드를 이용한 지원은 오버헤드가 너무 큼{**5장**})이 필요, OS가 정책에 따라 관리

**메모리 보호** (초기형) : **Base, Limit Register 사용**

-방법 : **프로세스마다** **독립된 주소 영역**(공간) 제공🡨유효하게 사용할 수 있는 범위(합법영역) 지정

- 프로세스의 합법적 주소 영역 = **Base ~ Base+Limit** register값 까지(base <= **address** <= base+limit)

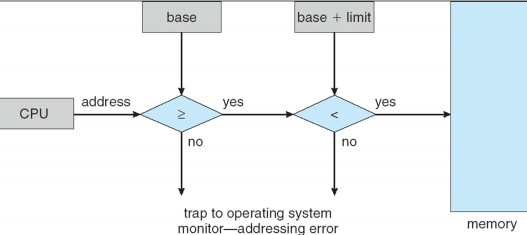
따라서 모든 프로세스에 base/limit을 지정해주어야 함

Bootstrap프로그램이 **OS커널을 적재(**주소0부터**)**하고 **init 프로세스(**첫 프로세스=부모 프로세스**)** 실행

🡪 **프로세스끼리 주소공간을 독립적으로 할당**함으로써 메모리의 서로 다른 공간 침해 방지

Base와 Limit register를 이용(하드웨어를 이용)한 주소검사

-> **합법이 아닌 주소(할당받지 않은) 접근에 대해서는 trap 발생(액세스 거부)** -> 메모리 보호

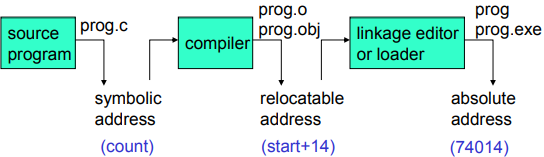


\* CPU가 발생한 **접근주소**가 Base보다 크면 통과/Base+limit보다 작으면 통과/아니면 trap발생

\* Base와 Limit register값은 **OS만 접근 가능한 특권명령어 사용(**OS는 메모리 전영역 사용가능**)**

**주소 바인딩** : 주소가 결정되는(실리는)것 **= 한 주소에서 다른 주소 공간으로 맵핑(사상)하는 것**

**\**논리적 주소 -> 물리적 주소***



소스프로그램의 상대주소(0~14) / 컴파일러에서 주소위치 재지정(+74000) / 링커, 로더를 통해 절대주소 적재

-**링커와 로더(바인딩 최종작업)**

- **링커** : 여러 모듈을 함께 묶어서 실행파일을 생성

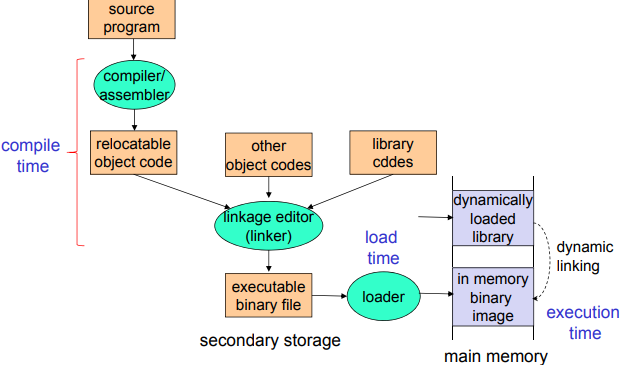
- **로더** : 실행을 위해 프로그램의 일부/전부를 메모리에 적재

EX) ABC.C(소스파일)---ABC.Obj------

라이브러리

ABC.C--------------ABC.Obj------ ---링커&로더---🡪**최종 실행 파일**

ABC.C--------------ABC.Obj------



\* Source program : AAA.c

\* Relocatable object code : AAA.Obj

**\* 링커를 통해 여러 모듈 묶어서 실행파일(.exe) 생성 🡪 로더를 통해 메모리에 적재**

**\*** library codes는 정적 라이브러리

\* dynamically loaded library : **프로세스가 library를 필요로 할 때 별도로 적재(동적)**

**\*** dynamic linking : **실행 중(실행시간)**에 linking이 일어남

주소 바인딩 시점 3가지 : 메모리 주소에 적재되는 시점

- 컴파일 시간 - 프로그램이 적재될 메모리 위치를 미리 알 때 컴파일러가 절대주소 생성

적재 시작 위치가 변경된다면, 코드를 다시 컴파일 해야 함

- 적재 시간 - 컴파일러가 relocatable code(맵핑{사상}된 주소) 생성 🡪 Load time에 주소위치 결정

- 실행 시간 – 프로세스가 실행 도중 메모리 주소위치 이동 가능 <- 바인딩이 실행 중 일어남

- 주소 맵핑을 위한 하드웨어 지원 필요 -> MMU(메모리 관리 장치)

\*MMU가 없으면 동적 바인딩 불가

- 대부분 OS가 이용

\*정적(Static) 바인딩 / 동적(Dynamic)바인딩

정적 바인딩 : 컴파일 시간, 적재 시간 바인딩

동적 바인딩 : 실행 중(실행시간) 바인딩<실행 중에 메모리 주소위치 변화>

**논리 주소**와 물리 주소

- **Logical address**(Virtual address) : 프로그램이 사용하는 주소 = 상대주소, ex) &a

- Physical address : 물리적 메모리장치가 사용하는 주소 = 절대(실제)주소

메모리 관리 장치(MMU) : **동적 바인딩에 필수적**

- 논리 주소를 물리 주소로 **맵핑하는 역할(프로세스 실행 중)**

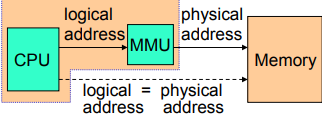
- 프로그램은 논리 주소만 사용 ,실제 물리주소를 알지 못함

실행 시간에

정적 바인딩(Compile/Load time) 🡪 **logical address = physical address**

동적 바인딩(Execution time) 🡪 **logical address != physical address**

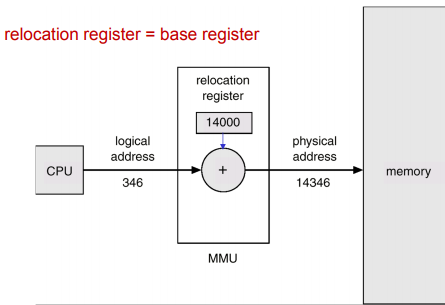
\***정적 바인딩은 애초에 프로세스를 메모리에 적재할 때 절대주소(실제주소)에 적재함**



\*CPU칩 안에 MMU가 포함되어있다. 🡪 따라서 CPU밖으로 나가는 신호는 모두 물리 주소

\*프로그램이 논리주소사용 -> MMU가 알아서 맵핑 -> 절대주소 사용

재배치 레지스터를 사용한 동적 재배치(바인딩), 초기형(Intel 80386에서 사용)

relocation레지스터가 +14000

**동적 적재**란? **실제 호출되기 전**에는 각 루틴은 **메모리에 적재되지 않음**

**호출될 때** 메모리에 없으면 relocatable 링커/로더를 호출 -> **메모리에** **적재 -> 실행**

**장점** : 메모리 공간 절약(사용치 않는 루틴은 적재X, 필요할 때만 적재)

많은 양의 코드가 잘 사용되지 않는 경우 효율 극대화(ex. 예외처리 코드)

\***동적 적재는 OS가 아닌 프로그래머가 설계를 하고 관리**, OS는 단지 동적 적재를 지원하는 Library만 제공(relocation register가 동적 적재하는 library 제공)

**동적 링크, 공유 라이브러리(**windows.dll, unix.so**)** : 동적으로 적재되는 라이브러리\*\*\*

- 정적 링크와 동적 링크

- 정적 링크 – 링커에 의해 Library가 실행 이미지에 링크

- *동적 링크* – Library 링크가 실행시간에 이루어짐

**Stub 코드** : 어떤 라이브러리를 찾아가야 하는지 탐색하는 코드 + 메모리에 library가 적재되어있는지 검사코드 <- 메모리에 library가 없으면 디스크에서 가져와서 적재(동적)

- 프로세스가 Library 함수를 호출 할 때 stub코드가 우선 실행 됨 🡪 stub 코드를 통해 library가있으면 탐색한 library 호출 / 없으면 디스크에서 library 가져와서 적재

- **stub**는 실제 **library함수 주소**를 **반환** 값으로 받아서(대체됨) -> 다음 호출부터는 library 함수 바로 호출하여 실행, 따라서 처음에만 stub코드가 실행되어서 library가져오면 이후 실행X

공유 라이브러리 : 동적 링크 라이브러리(.dll)

- 공유 라이브러리를 사용하는 모든 프로세스는 한 개의 library코드만 공유하여 사용

-> 라이브러리 업데이트 용이

\*동적 적재와 동적 링킹은 OS의 도움이 필요하다.

공유 라이브러리 추가 설명 (그림으로 그리기 P.12 필기)

스와핑 : **Memory 🡨🡪 Disk**

- 메모리의 저장 여유 공간이 부족 할 때 새로운 프로세스 실행을 위하여 메모리에서 실행 중이 아닌 프로세스를 디스크로 내보내고(Swap out), 새 프로세스를 디스크로부터 메모리에 불러오는(Swap in) 것

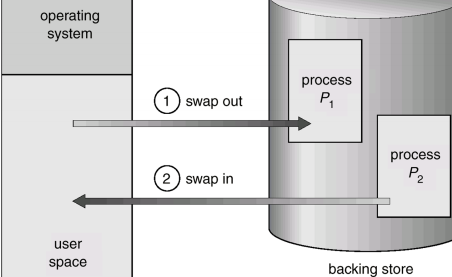
- Disk(예비 저장장치) : **모든 프로세스의 메모리 image의 복제본을 저장 + 빠른 디스크 사용**

-> 빠른 접근을 위해 **직접 접근이** 가능해야 함.

-> 빠른 접근법 : **File Sys을 거치지 않고 Disk 직접 액세스**

\*효과 : 물리적인 메모리의 크기보다 더 많은 크기의 프로세스를 동시 실행할 수 있음

Ex) **RAM(8GB) < 실행 프로세스 용량(9GB) -> 가능** (디스크 가상화 사용)



스와핑과 Context Switching 시간의 관계

스와핑 동작 순서

- **CPU 스케쥴러가** Ready queue에서 **다음 프로세스 결정**하고 Dispatcher 호출

- Dispatcher는 다음 실행할 프로세스가 메모리(RAM)에 있는지 또는 보조저장장치에 있는지 확인

-> 메모리에 없다 -> 적재할 메모리 여유공간도 없다 -> 메모리의 한 프로세스를 Swap out,

다음 실행 할 프로세스를 디스크에서 Swap in해서 수행(**Swap out후에 Swap in)**

- Dispatcher는 Context switching 수행

Context switch 시간

- Context switch time > **(Swap in + Swap out)** time

- **Swap 시간의 대부분이 디스크 전송시간**

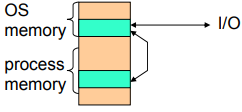
-> 시간 감소 방안(대책) : 프로세스에서 **실제로 사용하는 부분만** 불러오기(**Swap in)**

스와핑과 입출력

- Swap out(메모리에서 디스크로 내보내지는)되는 프로세스는 완전히 Idle(실행X)상태이어야함.

- 진행중인 I/O는 건들지 않고 스와핑하지 않음

-> I/O는 스왑하기 전 OS를 거침으로서 안정성을 높인다.(Double buffering)



**기본 스와핑<**현대 OS는 사용치 않는 방식**>**

- **RR스케쥴링** **: Quantum이 종료된 프로세스를 Swap out** =Time Quantum 종료 시 큐의 제일

뒤로 빠지게 되는데 큐의 제일 뒤 프로세스를 Swap out함

- **우선순위 스케쥴링** : **낮은 우선순위 프로세스를 Swap out**, 높은 우선순위 프로세스들에게

할당

**변형된 스와핑 <오버헤드 존재>**

- 많은 시스템에서 널리 사용 - Unix, Linux, Windows

- **평상시 Swapping 비활성화**(스와핑 자체가 오버헤드이기 때문에 **비활성함으로써 필요치 않**

**은 오버헤드 제거)**

- 여유 메모리 양이 *Threshold(일정기준)이하가 되면 Swapping 활성화 <-기준은 OS 정책*

- 여유 메모리 양이 증가하면 다시 Swapping 비활성화

또다른 스와핑

- 프로세스의 **일부만 스와핑** -> 스와핑 시간 감소 = 오버헤드 감소 = 성능 향상

모바일 SYSTEM에서의 스와핑

- **모바일 시스템은 스와핑 지원X**

-> 이유1 : 휴대폰은 영구저장장치로 HDD가 아닌 플래시 메모리(용량이 충분치 않음) 사용

-> 이유2 : 플래시 메모리의 쓰기 횟수에 대한 부담(플래시 메모리의 쓰기 약10만회 제약)

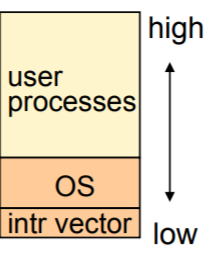
-----------스와핑을 사용치 않는 대신 사용하는 방법----------->>

**IOS** : 여유 메모리가 작아지면 응용프로그램의 메모리를 반환 -> 읽기 전용 데이터만 제거 -->>메모리를 **저장치 않고 종료하므로 응용프로그램을 시작하려면 처음부터 재시작**

**Android** : 여유 메모리가 작아지면 응용 프로그램 상태를 플래시 메모리에 저장하고, 프로세스 종료 -> **메모리가 저장된 후 종료되어서 재시작시 종료 직전 상태에서 재시작**

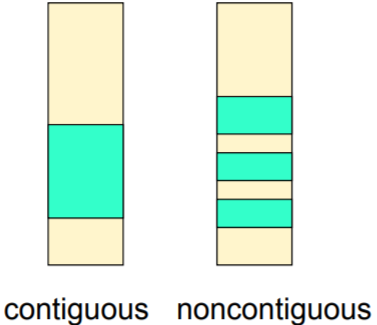
**메모리 할당**

주 메모리 : OS부분(메모리의 하위) / User프로세스 부분(메모리의 상위)

 \*intr vector는 Interrupt handler의 주소를 저장

**\*\***메모리 할당 방법 - **연속 메모리 할당** : 프로세스 **하나를 통째로 할당**

- ***비연속* 메모리 할당** : 프로세스를 나눠진 **메모리 공간에 분할하여 할당**

****\*contiguous=연속/ noncontiguous=비연속

재배치 레지스터(Relocation register)

- 메모리 관리 장치(MMU) => **재배치 레지스터 + 상한 레지스터** 이용

- **재배치(Relocation) 레지스터** : 프로세스가 배치된 물리적 주소의 **시작 주소** 저장

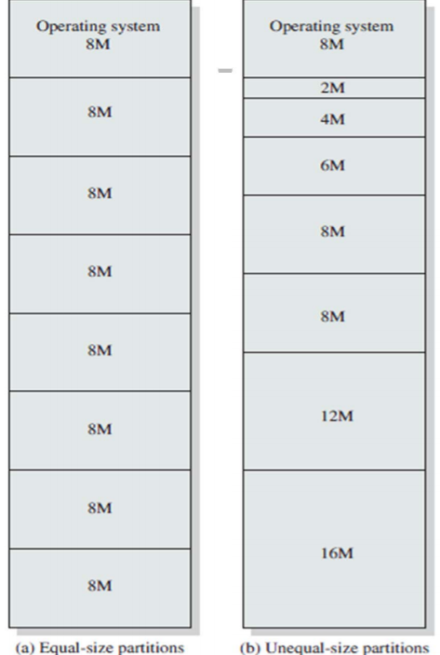
- **상한(Limit) 레지스터** : 논리주소의 **범위 지정**

-> 프로세스가 할당받는 영역 = Relocation 레지 값 **~** Relocation 레지+Limit 레지 값

\*메모리 보호를 위해 맵핑 사용

<- 각 **프로세스**가 사용하는 메모리 **주소 범위를 합법적 범위로 제한**, -> 프로세스가 다른 프로세스나 OS를 수정하는 것을 방지

\***MMU**는 **동적**으로 논리주소 -> 물리주소 맵핑

연속 메모리 할당의 **2**가지 방식

**1.** **고정 분할 방식** : 메모리를 몇 개의 고정 크기로 분할

- 동일 크기 / 다른 크기 분할

-> **분할된 파트(=hole)마다 1개의 프로세스만 포함가능**

-> **OS측**에서 관리는 쉽지만, / 단점 : 분할된 메모리의 **한칸 (Part)보다** 큰 프로세스 할당 불가

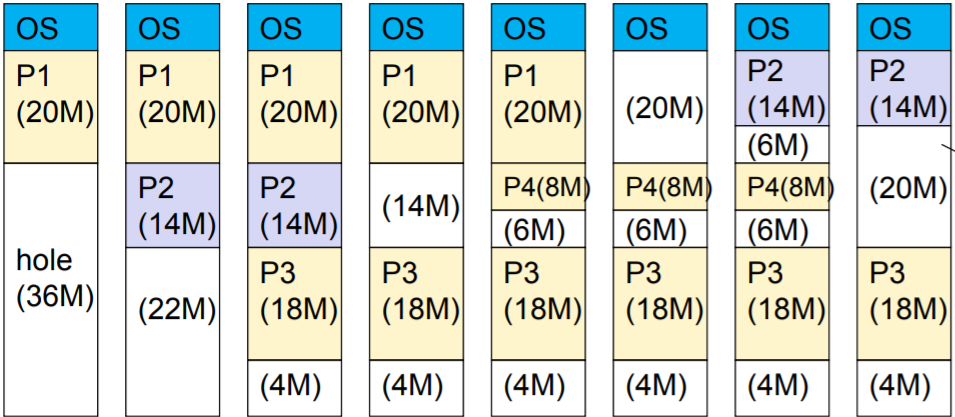
-> 파티션 갯수 >= 멀티 프로그래밍 정도(갯수)

동일 크기 / 다른 크기 (고정분할)

**2. 가변 분할 방식 :** 필요한 만큼 가변적 할당 => 효율적인 메모리 관리 가능, / 관리가 어려움

- OS의 관리영역 1) 메모리의 **할당된 파티션** 2) 사용 가능한 **빈 파티션**

-> 프로세스를 수용할 수 있는 크기의 hole에서 메모리 할당



\* 가변 분할 방식(크기에 딱 맞춰서 메모리 할당) / **8M 프로세스**를 실행(할당)하려할때 메모리 **전체의 빈 공간은 6+4M=10M**가 여유지만 연속 메모리할당 방식에서는 **할당 불가**

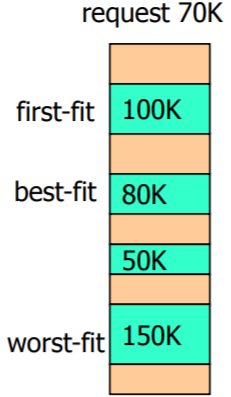
\* Hole = Free partition

동적(Execute time : 실행시간) 메모리 할당 **방식**(연속 메모리 할당에서)

1. **First fit** : 프로세스를 수용할 수 있는 **첫 hole(free partition)에 할당**

-> 검색 시작 위치부터 리스트 탐색 -> 탐색 중 처음으로 할당 가능한 크기에 할당

2. **Best fit** : 프로세스를 수용할 수 있는 **가장 작은 hole에 할당** = **최적 Fit** hole에 할당

 -> 리스트가 정렬 되어있지 않으면 모든 리스트를 탐색해야 함

-> **할당한 결과 가장 작은 크기의 hole이 생성됨**

3. **Worst fit :** 요청을 수용할 수 있는 가장 큰 hole에 할당

-> 리스트가 정렬 되어있지 않으면 모든 리스트를 탐색해야 함

**메모리 단편화 : 메모리 낭비로 인해서 사용치 못하는 메모리 영역 발생**

메모리 단편화 종류(내/외)

- ***외부 단편화*** : 외부에서 요청할 때 **요청한 크기보다 작은 메모리 블록만 존재**

**-> 결과 :** 요청한 프로세스를 할당 할 수 없음

**- *내부 단편화*** : 요청 받아온 프로세스를 메모리로 할당할 때 **더 큰 메모리part를 할당**

**-> 결과 :** 할당된 메모리 일부가 사용되지 않음(비효율성)

-> **고정 분할 방식**에서 발생(메모리를 미리 분할해놔서 프로세스 크기에 딱 맞기 힘듦)

\*외부 단편화 : OS가 인식함 / 내부 단편화 : OS가 인식 못함

단편화 **문제**

-외부 단편화 문제 : **50퍼센트 룰** : N개의 블록이 할당 되면 0.5N개 블록이 외부 단편화로 사용 불가 --> 0.5/1.5 낭비 = 1/3 낭비 -> 해결을 위한 hole 추적 시 오버헤드가 매우 큼

-내부 단편화 문제 : 할당된 메모리의 일부는 사용되지 않음 = 낭비

압축(Compaction)

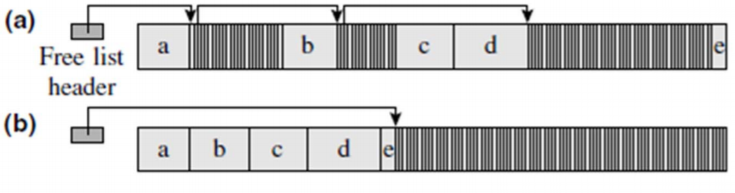
외부 단편화 문제 **해결**책 2가지

1. 압축(compaction) 2. 비연속 물리 메모리 공간 허용/이용(**Page방식**, segmentation방식)

압축

- **모든 free 메모리 공간들을 하나의 큰 블록으로 재배치** => 시간 소요多, 동적이어야 가능

- 모든 프로세스가 동적이어서 실행시간에 수행가능할 때 가능

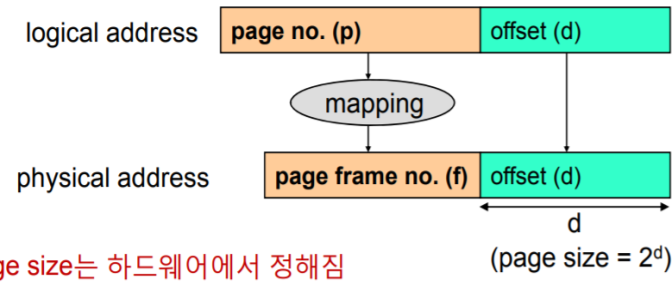


**페이징 : 비연속 물리적 메모리 할당 방식**(할당할 1개의 프로세스 크기만큼의 연속된 메모리 공간이 없을 때 메모리에 *분할되어있는* ***free partition(hole)****들의 크기로 프로세스를 분할하여(나누어) 각각 맵핑*)

- Page frame : 분할되어있는 물리적 메모리를 각각 고정된 크기의 블록으로 나눔(512B~4KB)

-> Page : 할당할 프로세스의 논리 메모리를 Page frame과 같은 크기로 각각 나눔

-> page를 page frame에 맵핑 <= Page와 Page frame 크기가 동일

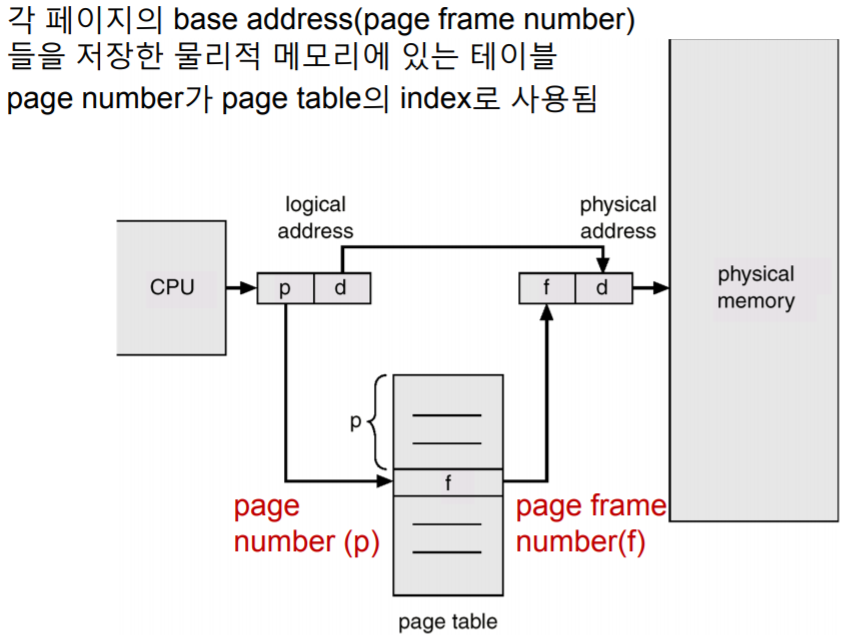


주소 변환 방식

- 상위 비트 : Page Number(페이지 번호)

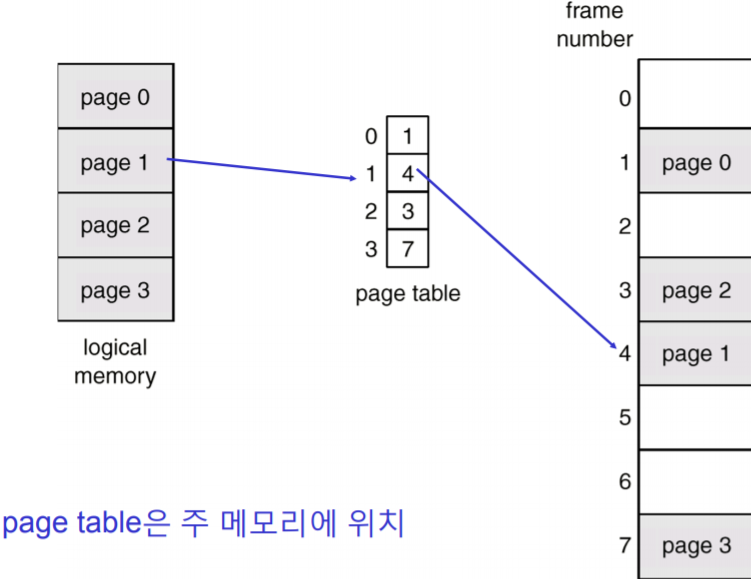
- 하위 비트 : 해당 Page번호에서의 Page offset

Page와 Page frame 블럭 크기가 동일



Page Table의 관리는 OS의 역할

Paging의 예시



page table은 주 메모리(RAM)에 위치하고 분할된 프로세스를 각 Page Frame에 맞게 사상(맵핑)하여 보냄

**프로세스마다 자신의 Page Table을 가지고 있어서** 프로세스끼리 사상 메모리영역이 겹치지 않음

**Page table은 몇번째 Page frame인지 찾아갈 수 있도록 Page frame number 정보 저장**

**<- 상위 비트로 물리주소의 Page Frame번호를 찾고 하위비트로 해당 Page Frame(=page)내에서의 offset(상대)주소를 찾아감**

Page 할당하는 법

- 메모리 상의 모든 Free Frame을 추적(탐색) -> Free Frame List 사용하여서

**- N** page 크기의 프로그램을 실행하려면 **N** free frame을 찾아서 **page에 1:1 할당** -> 프로그램 적재

- 주소 변환을 위해 Page table을 할당된 Page frame 번호들로 설정

*내부 단편화가 발생할 수 있음(소주의 막잔이 부족한 것 처럼)*

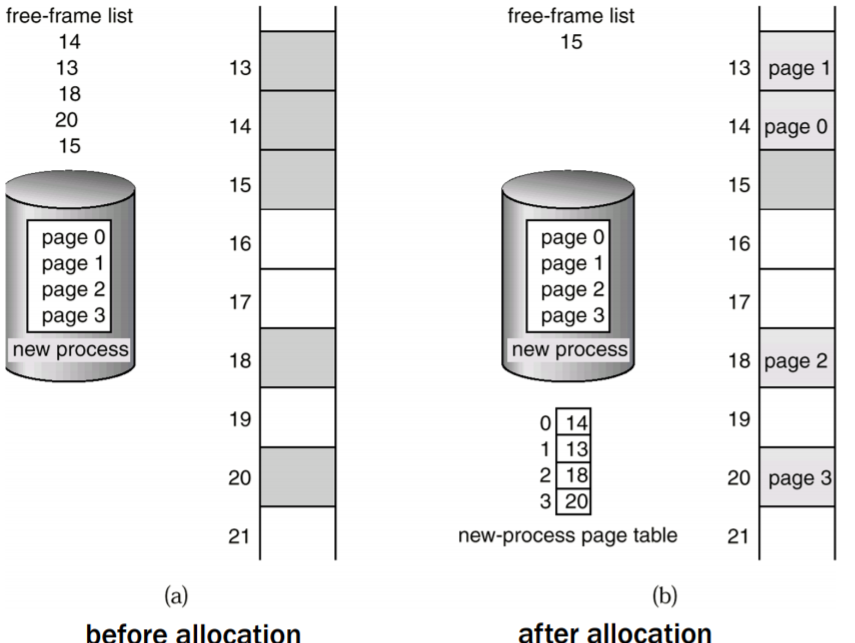
- 메모리 할당 단위 : Page frame 단위(고정) -> **내부단편화**(요청받은 프로세스 메모리보다 더 큰 메모리공간 할당) 발생, *외부 단편화X* <- **Page frame에 딱 맞게 page와 맞춰서 할당되지만 분할된 마지막 page는 Page frame(고정)보다 작을 수 있음** <- 내부단편화가 생기는 이유

\*페이지 단위를 **작게하면** 내부단편화 된 메모리가 **작아진다**. + 페이지수 **증가**(관리 어렵)

\*페이지 단위를 **크게하면** 내부단편화 된 메모리가 **커진다**. + 페이지수 **감소**(관리 쉽)

\*따라서 복수의 페이지 크기 지원 **: OS커널** - Solaris : 8KB, 4MB / **CPU** - Intel Pentium : 4KB, 4MB

\***Free frame list**가 있고 새 프로세스 적재시 list에서 **순서대로 맵핑** Func가져와서 적재



**Frame table 과 Page table**

-Frame table : Page frame의 정보를 저장, 어느 프로세스의 어느 페이지에서 할당되었는지 기록(역방향)

-Page table : **OS는 각 프로세스마다 자신의 Page table을 가진다**.

-> Context Switching(프로세스 교체)시에 **Page table도 백업**해야 하므로 Context Switching 시간 증가 **<Page table도 PCB정보 중 하나>**

Page table(**한프로세스가 분할되어 메모리에 적재되어있는 Page frame들의 주소정보를 저장)**의 구조

1. Page table 전용 레지스터 집합에 위치(Table이 아주 작을 때만 가능, 따라서 거의 사용X)

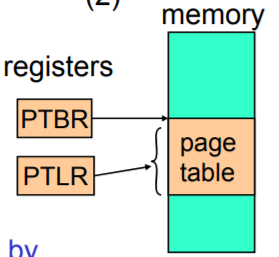
2. **Page table을 주 메모리에 위치(**많이 사용)

- **PTBR**(Page Table Base Register) : page table의 **시작 주소** 저장(프로세스마다 각자의 시작주소 존재)

- **PTLR**(Page Table Length Register) : Page table의 **크기(범위),**base ~ base + length = 합법주소

<<-- Page table의 시작주소와 page table의 주소크기(범위, 길이)값을 가지고 있음, **레지스터 값을 통해서 Page table을 찾아감.**

**<<--OS가 특권 명령어를 사용하여서 관리**

(분할되어 적재되어있는 프로세스 Page frame 위치정보를 가짐)

\*\*\* -> **프로세스당 각자의 Page table을 가지므로 Context Switching시 CPU칩 내부의 레지스터인 PTBR, PTLR의 값을 바꿔줌으로써 다른 프로세스의 Page table을 가리키게 한다.**

\*캐시와 레지스터는 모두 **CPU칩 내부**에 존재

메모리에 있는 Page table과 TLB(캐시)

- 모든 Data/Instruction 접근에 2번의 메모리 접근이 필요함

-> 레지스터를 통해 **Page table에 1번** 접근 + table을 통해 해당 **Page frame에 1번 접근**

**-> 실제로는 메모리를 2번 접근할 수 없음(불가능)**

- **TLB** : Translation Look-aside Buffer캐시(Page table의 정보(내용)을 저장)

-> 메모리 2회 접근을 **해결하기** 위해 Page table entry용 **캐시** 사용

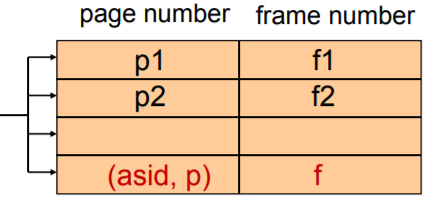
-> 주소 변환 정보(Page table)를 CPU내 캐시에 두고 사용

-> 가장 빠른 방식의 **완전 연관 사상 캐시 사용(Hit Ratio가 가장 높다)**

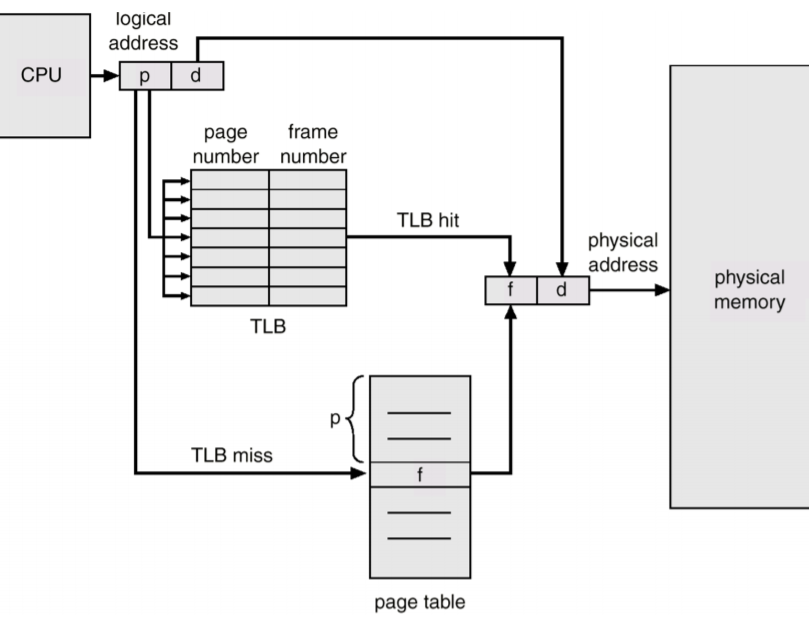
-----문제점 : 새로운 Page table을 사용(프로세스 교체 = Context Switching)할 때 마다 TLB내용은 flush(비움, 무효화)되어야 함

--->> **Address Space Identifier(ASIDs)를 함께 저장하여 flush를 필요없게 함 \*\*?**

**<**프로세스 교체 시 마다 새 Page table이 필요한데, 그 이유는 각 프로세스는 각자의 독립적인 Page table을 가지고 있으므로 Page table 내용을 갖고있는 TLB내용 또한 바뀌어야 한다. 따라서 TLB가 Flush되어야 하고 가장 히트율(Hit Ratio)가 높은 완전 연관 사상 방식 캐시(교체정책이 복잡)를 사용한다.>

\* TLB가 Full로 사용 중이면 OS는 새로운 entry정보 저장을 위해 교체 대상 entry정보를 선택하고 교체한다(방식 : LRU, FIFO, RANDOM, RFU 등등)

Paging hardware with TLB 그림



CPU에서 주소신호 발생

->1. TLB 캐시를 먼저 검사

->2-1. 히트시, Page frame NO(번호)를 얻어서 이동

->2-1. Miss시, Page table(주메모리에 위치)을 탐색해서 TLB로 Swapping, 적재+Page frame NO로 이동

TLB는 CPU내, Page table은 메모리내

메모리 보호

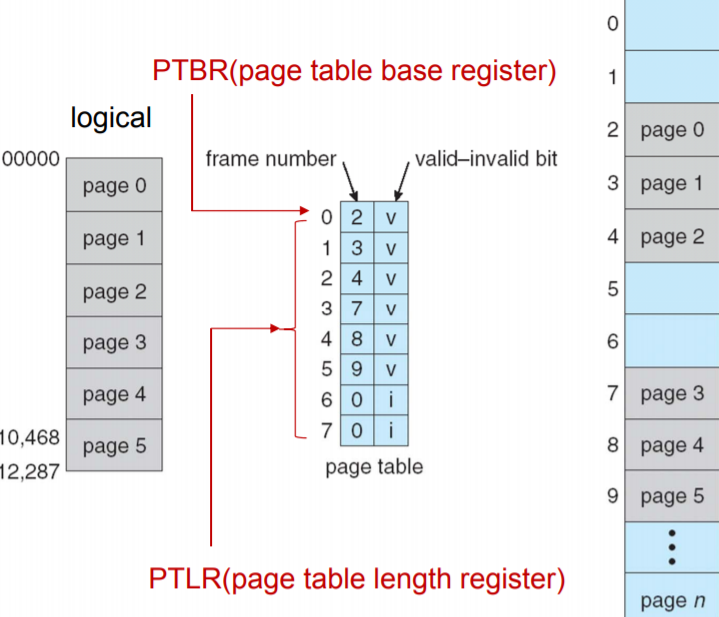
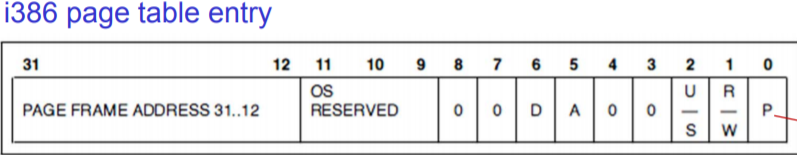
-**Page 단위**에서 메모리 보호를 위해서 **page table entry**에 Protection bit와 Valid bit 추가

-Protection bit : Page 단위 보호 제공(Read Write, Read only, Executable only을 통해서)

*읽기+쓰기, 읽기 전용, 실행 전용(읽기 조차안됨)*

-Valid bit : V=1 ->프로세스의 페이지들(번호)이 Page table에 있음

V=0 -> Physical 메모리에 페이지가 없음



페이징의 장점 : **Code를 쉽게 공유**할 수 있게 해준다 -> **Multitasking 구현**

페이징이란? 프로그램(프로세스)에게 실제로는 물리적 메모리 공간에 페이지 단위로 분할되어

적재되어 있지만 한 프로그램의 주소가 쭉 연속되어 있는것처럼 환상을 심어주는것

-공유 코드 : **재진입 코드는** 공유(여러 프로세스가 공유) 될 수 있다.

- **재진입 코드는** 수행 동안 변하지 않는 코드(**읽기 전용**) <- 명령어부분(코드)

- 따라서 읽기 전용인 재진입 코드(공유 코드)는 한개의 **코드가 여러 프로세스에 공유**되고 /

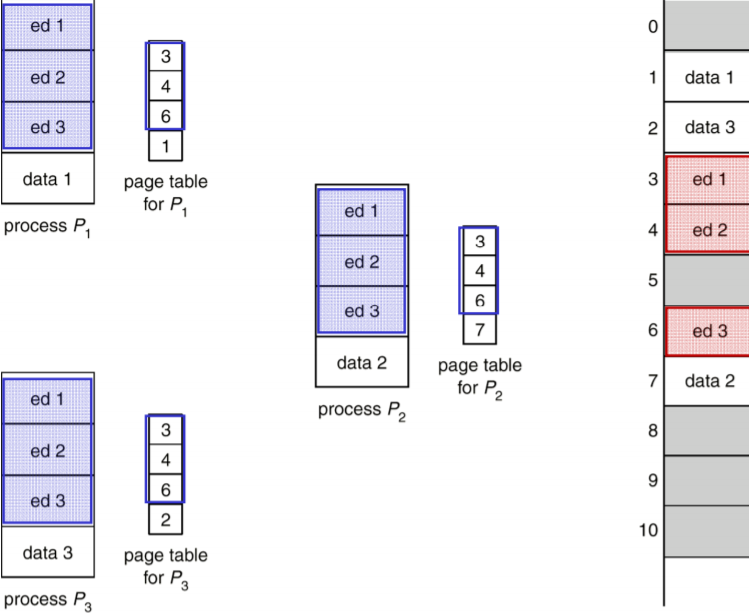
**데이터는 프로세스마다 각자의 저장공간** 사용

\***공유 코드는 모든 프로세스의 논리 주소 공간에서 같은 위치에 있어야 한다.**

<- 서로 다른 프로그램이 코드를 공유할 수 있기 때문에 논리 주소 공간을 같은 위치로 맞춰줘야 함. **공유 코드의 예시 : 공유 라이브러리(**앞장에 설명)

**TIP. 서로 다른 프로세서들이 코드를 공유하기도 한다.(공유 Library)**

Paging을 통한 코드 공유 그림(코드<명령어>는 물리적 공간에서 공유, 데이터는 프로세스마다 따로 사용)



**Page table 구조**

-Page table의 크기

EX. 32비트 주소공간, 1페이지 = 4KB 크기

- 2^32<4GB>/2^12<4KB> = 1M Page table entries = Page frame 1M개 존재

\* Page table에 Page frame들의 번호 및 정보들 저장

최신 컴퓨터는 ~2^64까지 주소공간 사용 -> Page table의 크기가 커진다.(page frame갯수 증가)

-> Page table의 크기(사이즈, 적재용량)가 한 Page 사이즈보다 더 커지게 된다.

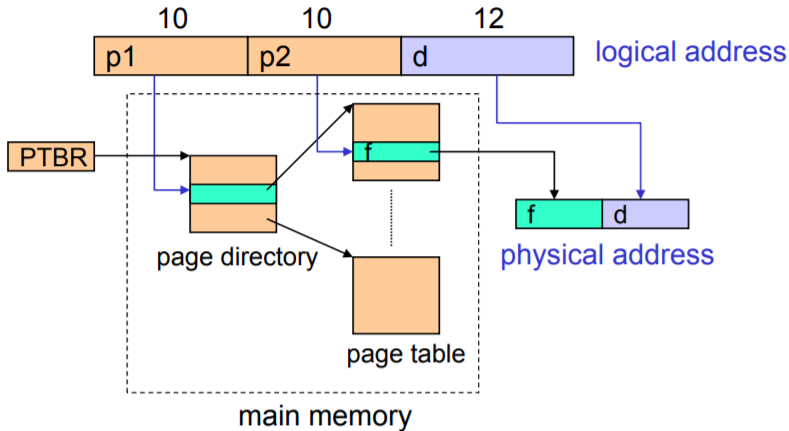
-해결책 : Hierarchical Paging(계층적 페이징), Hashed Page tables(해시함수 사용)

**계층적 페이지 테이블**

- 다단계 page table

- **Page table을** 하나의 연속적 메모리로 사용하지 않고 **여러 개의 작은 조각(1page 크기로)**

**으로 나눠서** 계층적 구현 -> Page table이 Paging된 개념, Page table을 분할



<-상위 20비트를 10비트씩 나눠서 page directory / page table로 사용

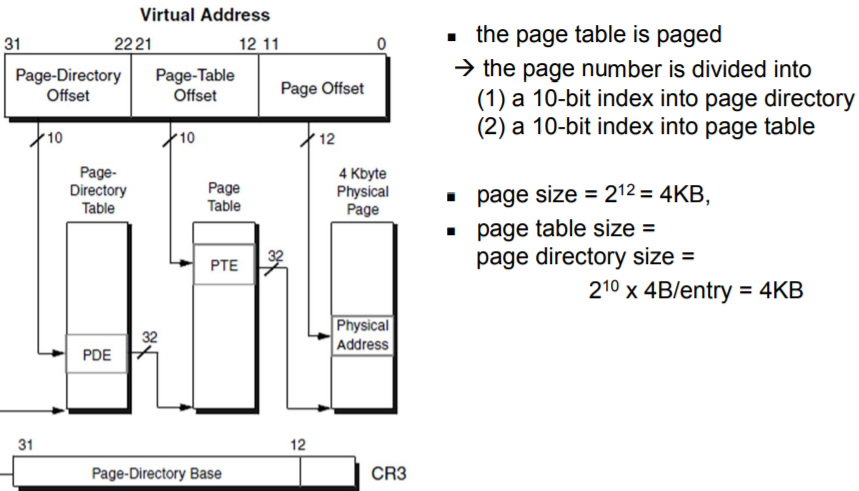
\*P1 : Page directory(몇번째 Page table인지 탐색) 탐색 주소

\*PTBR : Page Table Buffer Register

\*P2 : P1에서 탐색된 Page table에서 몇번째 Page frame인지 탐색

\*d : Page frame(Page)에서 논리 주소(Offset, 상대주소) , 12bit이므로 2^12 = 4KB(1page크기)

-80386 예시



\* 더 많은 page table공간을 쓸 수 있다.

\* 2단계가 아닌 **3단계** Paging(P1,P2,P3)을 구현하면 TLB에 정보가 없을 때 Page table을 찾아서 Page까지 도달하는데 **메모리 액세스가 4번이나 발생**하게 됨 -> 오버헤드가 너무 커짐(64bit에서)

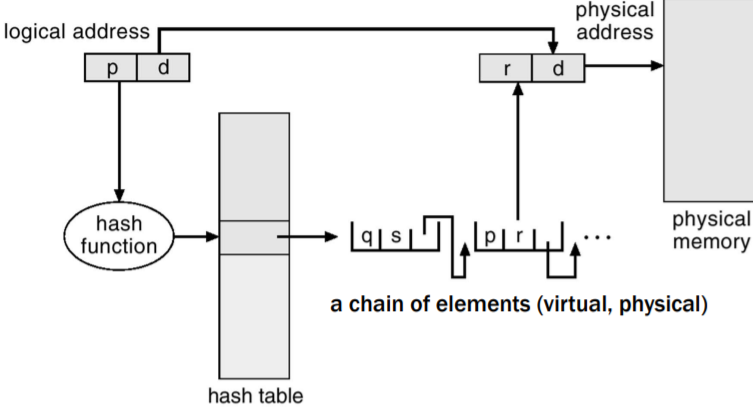
64bit에서는 Hierarchical page table이 가능은 하지만 오버헤드도 크고 비효율적이어서 **부적합**

-> Hash Func을 이용한 **Hash Page Tables** 사용

-> 현대 컴퓨터 시스템은 거의 64비트를 사용하므로 Hash Page Tables를 사용한다고 보면됨

**Hash Page Tables**

- 주소공간이 32비트(4GB){<- 32비트 운영체제는 4GB까지밖에 인식 못함}보다 클 때, Hash 함수를 Hash page table의 배열(index)로 사용



Linked list를 사용하여 구현(탐색)

linked list로 검색시 운좋으면 한번에 탐색, 나쁘면 마지막 리스트까지 이용

V, P

p : 상위비트

d : 하위비트(Page frame에서 상대주소)

해시 함수(Ex. mod X)

줄어든 검색 범위

(함수로 인해 여러범위 발생 가능)

\*Hash Func(함수, Ex. **mod X**)를 사용하여 나온 값을 Hash table***의 검색 대상 범위를 줄일 수 있음*** + **Hash 함수에 의해 여러 범위의 주소영역이 선택될 수 있다.(**mod X 사용시,**hash table범위/X개**)

Clustered page table은 참고만하고 건너 뜀

**Segmentation**

- 프로그램은 세그먼트의 집합이다. <- 사용자 관점의 메모리

-세그먼트는 프로그램의 논리적 단위 : 고정된 크기로 분할하는 것이 아닌 논리적 단위 분할

-> **논리적 단위**란 : Main, 프로시져, 스택, 배열, Local 변수, Global 변수 등등

-> **논리적 단위는 분할된 크기가 서로 다를 수 있다.**

**Segmentation의 논리 주소** = (**segment number, 해당 segment 내에서 offset**) = **2차원** 주소

-> 실제 물리적 메모리는 1차원 주소를 갖기 때문에 2차원을 1차원으로 맵핑

-> Row, Column 맵핑

**Segment table 존재** <- Page table과 같은 맥락

- 프로그램의 각 세그먼트의 주소 변환정보 저장

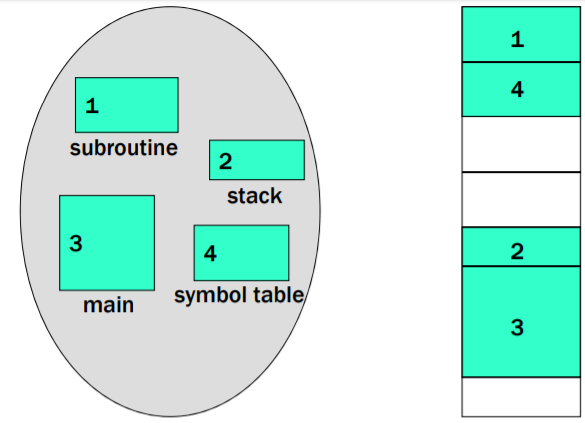
- Segment table entry 형식 : **Base - 세그먼트의 시작 물리주소**

**Limit - 세그먼트의 길이**

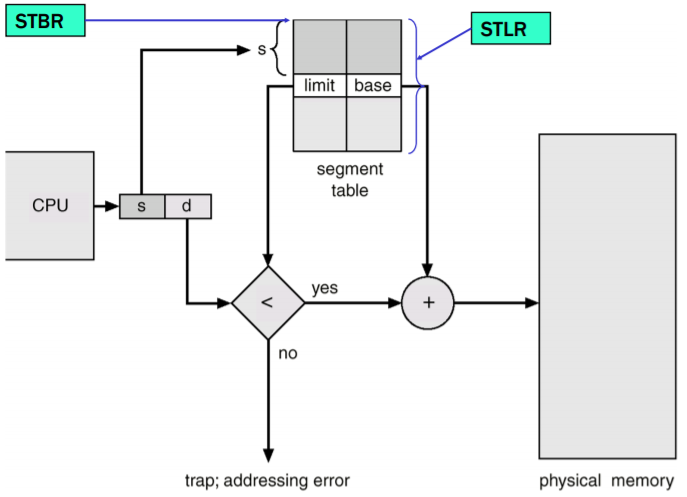
- Segment 길이는 **가변** -> 메모리 할당도 **동적으로** 이루어짐

- STBR : Segment Table Base Register

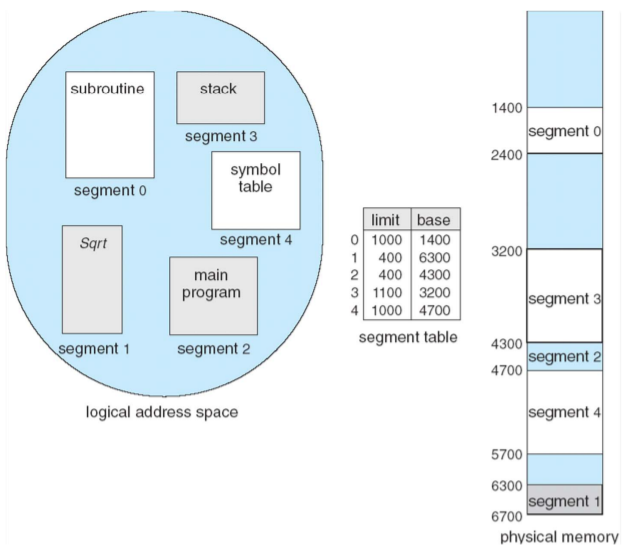
- STLR : Segment Table Length Register

왼:1개의 프로세스, 오:물리적 메모리

Segmentation(프로그램 코드의 논리적 분할단위) 하드웨어와 주소 변환



\* 합법적 주소가 아닐 시, Trap 발생 \* 잘못된 접근을 막을 수 있음

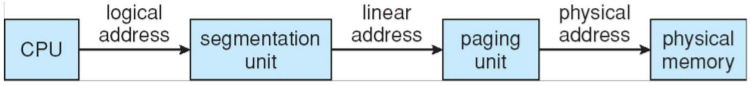
**\* 일반적으로 1 세그먼트는 1 페이지보다 단위가 더 크다.**

<참고만>

Segmentation의 예 : Pentium계열

- Segmentation과 Segmentation with paging을 지원

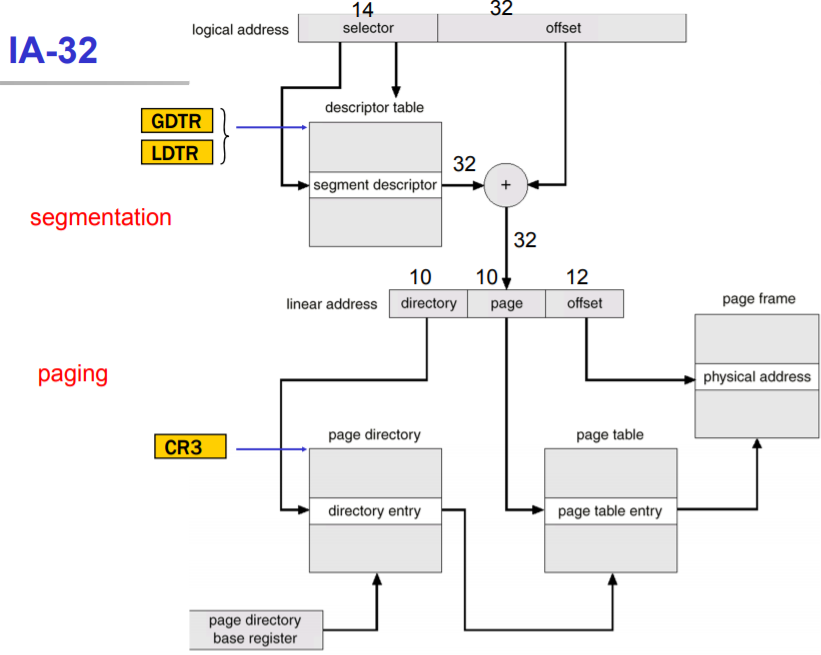
- 주소 변환 과정



CPU에서 **논리주소(2차원) 발생**-Seg unit에서 **1차원주소로** 맵핑-상대적 주소를 **절대주소(M)로** 변환

<- Paging을 사용하지 않으면 1차원 선형 주소가 바로 물리적(절대주소) 메모리 주소가 됨

IA-32(Pentium) Seg : offset = 14 : 32



\*강의자료 프린트 p61 참고 + 이해만 하고 넘어가기 (pentium, ARM{2단계 페이징})

