# 3. Operation System ( Memory & Cache )

https://myvelop.tistory.com/201

https://dapsu-startup.tistory.com/entry/운영체제Operating-System-메모리Memory

https://technote-mezza.tistory.com/92

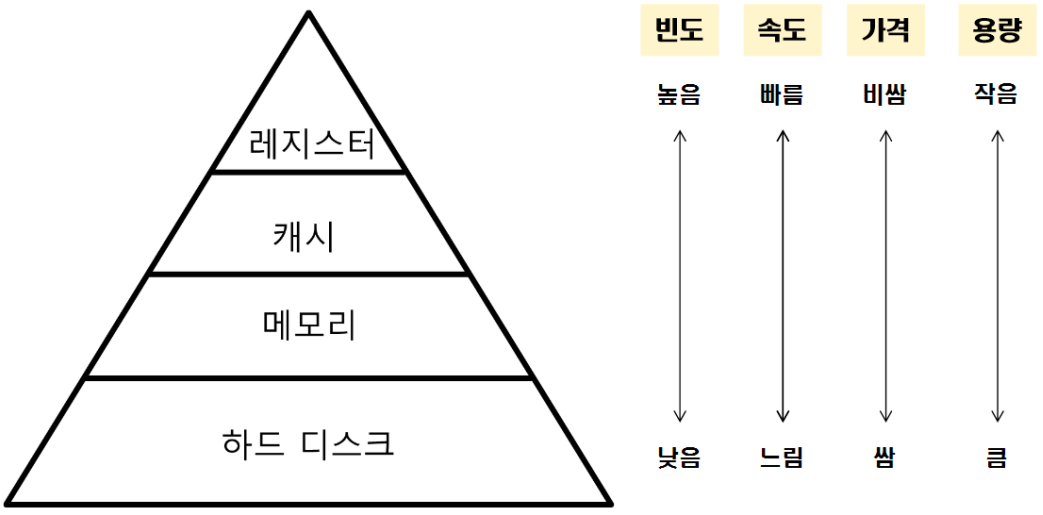
---

## 1. 메모리 기본 원리 & 메모리 시스템

### ※ 메모리란?

⇒ 메인 메모리, RAM을 뜻한다. 프로그램 실행 시 필요한 주소, 정보들을 저장하고 가져다 사용할 수 있게 만드는 공간 ( 작업을 위해 사용되는 공간 )

⇒ 메모리 시스템은 여러 가지 용량, 비용, 접근 시간을 가지는 저장 장치들의 계층구조




- **CPU 레지스터**: 가장 자주 이용하는 데이터 보관( locality ), CPU 내의 연산을 위한 저장 장치

- **캐시**: SRAM 라고도 하며, CPU 부근에서 비교적 느린 메인메모리에 저장된 데이터와 인스트럭션들의 부분집합에 대한 준비장소
  - 매번 메인 메모리에 직접 접근하는 것은 비효율적, CPU와 메인 메모리 **속도 동기화**!
  - 정적 램이라 불리며, 동적 램보다 더 빠르며 훨씬 비쌈, 캐시 메모리 → CPU 칩 내부 / 외부
  - 데스크톱 시스템에서 수십메가바이트의 SRAM을 가진다.
  - 각 비트를 이중안정 메모리 셀에 저장, 때문에 자신의 값을 전력이 공급되는 한 무한히 유지
  - DRAM과 달리 리프레시가 필요 없으며, 외부 교란에 민감 X
  - DRAM보다 트랜지스터를 더 많이 사용하며, 더 낮은 밀도와 함께, 전력 ↑ & 더 비쌈

**※ 캐시의 종류**

1. **L1 Cache** : 프로세서와 가장 가까운 캐시, 속도를 위해 I$와 D$로 나뉜다

a. **Instruction Cache(I$)** : 메모리의 TEXT영역 데이터를 다루는 캐시

b. **Data Cache(D$)** : TEXT영역을 제외한 모든 데이터를 다루는 캐시

2. **L2 Cache** : 용량이 큰 캐시, 크기를 위해 L1캐시 처럼 나누지 않는다.
3. **L3 Cache** : 멀티 코어 시스템에서 여러 코어가 공유하는 캐시

**※ 캐시의 성능**

1. Hit latency : 요청한 데이터가 캐시에 존재하는 경우, 히트가 발생해 캐싱된 데이터를 가져오는 시간
2. Miss latency : 요청한 데이터가 캐시에 존재하지 않는 경우, 미스가 발생해 상위 캐시 또는 메인메모리에서 데이터를 가져오는 시간

   ⇒ 평균 접근시간은 **Hit latency + (Miss rate * Miss latency)**

   ⇒ miss rate를 줄인다면 비용 감소,  캐시의 효율을 극대화 → 지역성의 원리!

**※ 캐시라인 ( cache line )**

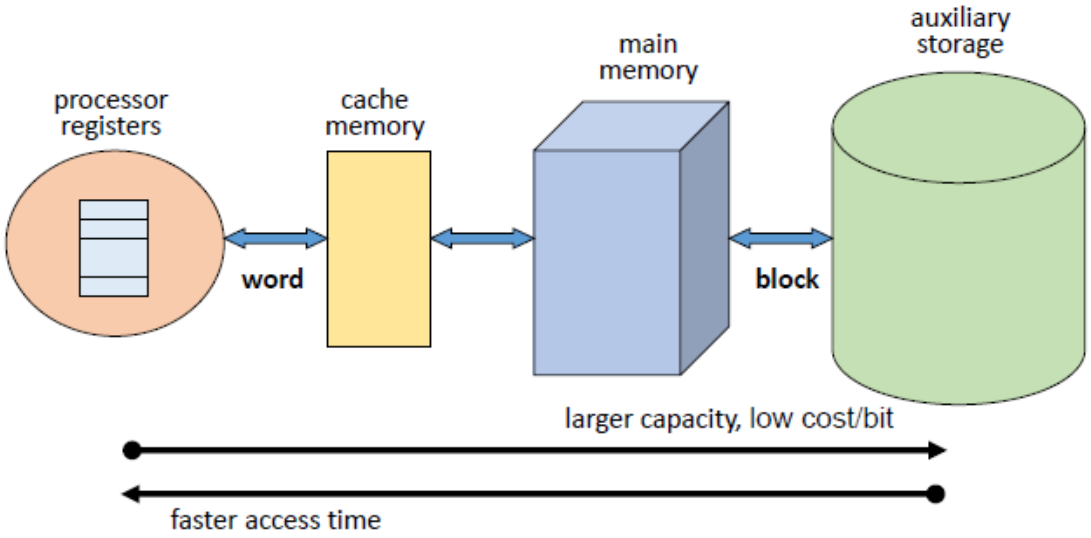
⇒ 캐시에 목적 데이터가 저장되어있을 때 바로 접근하여 출력하도록 도와주는 자료구조

⇒ set이나 map 등의 자료 구조를 활용, 캐시에 데이터 저장, 데이터 & 데이터의 메모리 주소의 쌍!

⇒ 캐시 = 하드웨어로 구현한 hash table, 또는 map과 같은 의미! 때문에 시간복잡도가 매우 빠름

- **메인 메모리**: DRAM 이라 하며, 크고 느린 디스크에 저장된 데이터를 준비하는데 사용, 이 디스크들은 네트워크로 연결된 다른 머신들의 디스크에 저장된 데이터를 위한 준비 장소
  - 메인메모리와 그래픽 시스템 간 프레임 버퍼, 수백 ~ 수천 메가바이트의 크기를 가진다.
  - 주기적으로 모든 비트를 읽었다가 다시 써 주는 리프레시를 처리한다.
  - 빛 또는 전기적 잡음 등의 외부 교란에 민감하다.

※ 단, SRAM & DRAM 둘 다 휘발성 성질을 가진다.




- **Block**
  - 보조기억장치와 주기억장치 사이의 데이터 전송 단위
  - Size: 1 ~ 4KB
- **Word**
  - 주기억장치와 레지스터 사이의 데이터 전송 단위
  - Size: 16 ~ 64 bits

## ※ 메모리 관리의 필요성?

⇒ 각각의 프로세스는 독립된 메모리 공간을 갖고, 운영체제 혹은 다른 프로세스의 메모리 공간에  접근할 수 없는 제한이 걸려있다.

⇒ 운영체제 만이 운영체제 메모리 영역과 사용자 메모리 영역의 접근에 제약을 받지 않기 때문에  운영체제에서 메모리를 관리한다.

⇒ 멀티프로그래밍 환경으로 변화하면서 한정된 메모리를 효율적으로 사용해야 했고, 운영체제가  이를 어떻게 관리하는지에 대한 관리방법이 중요..

### ※ 운영체제의 메모리관리의 역할

⇒ 실행파일이 로더에 의해 메모리에 올라오고 운영체제는 이 실행파일을 메모리에 어느 부분에  올릴지 결정한다.

⇒ 자바는 OS의 메모리 영역에 직접 접근하지 않고 JVM를 이용해서 간접적으로 접근한다 .

## ※ 동적 로딩 ( Dynamic Loading )

⇒ 해당 루틴이 불려질 때 마다 메모리에 load하는 것, 메모리의 이용성(Utilization)을 향상시킴!

⇒ 프로세스 전체를 메모리에 미리 다 올리는 것이 아님, 많은 양의 데이터가 한꺼번에 메모리에 Load되는 것을 막아 좀 더 효율적으로 사용할 수 있도록 함

⇒ 이를 통해, 기존 방식보다 메모리에 더 많은 프로그램을 적재할 수 있도록 한다.

⇒ 운영체제 도움 없이, 프로그램 자체에서 구현 가능한 성질

## ※ 동적 연결 ( Dynamic Linking )

⇒ 연결 ( Linking ) : 프로그래머가 작성한 소스 코드를 컴파일하여 생성된 목적 파일( object file )과, 이미 컴파일된 라이브러리 파일들을 묶어 하나의 실행 파일을 생성하는 과정을 말한다.

⇒ 동적 연결 : 컴파일을 통해 생성된 목적 파일과 라이브러리 파일 사이의 연결을 프로그램의 실행 시점까지 지연시키는 기법, 라이브러리 실행 시점에 연결을 수행한다.

⇒ 정적 연결 ( Static Linking ) : 작성 코드와 라이브러리 코드가 모두 합쳐저 실행 파일을 생성하며, 파일의 크기가 상대적으로 크다.

⇒ 실행 파일에 라이브러리를 포함 X, 프로그램 실행 → 라이브러리 함수를 호출함으로 서로 연결!

⇒ 실행 파일의 라이브러리 호출 부분 → 라이브러리 루틴 위치를 찾기 위한 stub 코드 설정

⇒ 메모리에 라이브러리 존재 → 해당 주소로 즉시 이동, 없으면 디스크에서 읽어온다.

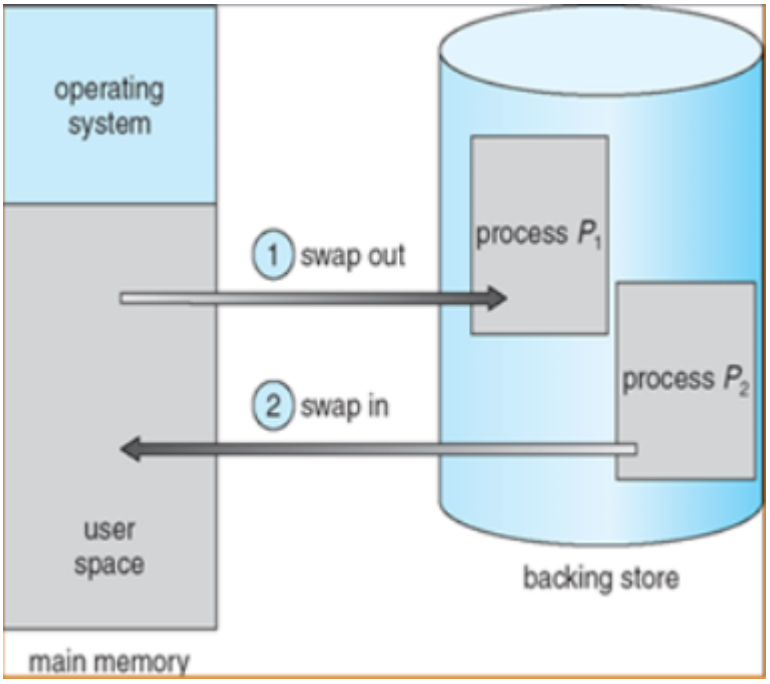
⇒ 메모리의 효율을 높일 수 있는 대신 운영체제의 도움이 필요하다.

## ※ 중첩 Overlays

⇒ Overlays : 프로세스 주소 공간 분할해 실제 필요한 부분만을 메모리에 적재하는 기법

⇒ 프로세스 크기가 메모리 사이즈보다 큰 경우, 당장 필요한 부분만을 메모리에 올려 실행한다.

## ※ Swapping




⇒ 메모리에 올라온 프로세스의 주소 공간 전체를 디스크의 Swap 영역에 일시적으로 내려놓는 기법

⇒ 메모리에 적재되어 있으나, 현재 사용하지 않는 프로세스를 관리하는 역할, swap에는 큰 디스크 전송 시간이 필요하기 때문에, 메모리 공간이 부족할 때 Swapping이 시작!

( swap time의 대부분을 차지하는 값은 transfer time, 디스크 전송시간 )

⇒ 디스크의 스왑 영역 = backing store, 디스크 내 파일 시스템과는 별도로 존재하는 공간

⇒ swap out : 디스크로 보내는 것, swap in : 디스크에서 메모리로 적재하는 것

⇒ 보통 스케줄링 환경 하 CPU 할당 시간이 끝난 프로세스의 메모리를 backing store로 보낸 뒤, 또 다른 프로세스의 메모리를 불러들인다.

⇒ 우선순위를 기준으로 낮은 프로세스를 swap out & 높은 프로세스를 메모리에 올린다.

⇒ “Swapper”라 불리는 중기 스케줄러에 의해 swap out 될 프로세스가 결정되며, Runtime binding 방법으로 주소를 할당하는 것이 swap in/out 할 때 좋다.

---

# 2. 주소 & 주소 바인딩

## 1. 주소

⇒ 서로 다른 위치를 구분하기 위해 사용하는 일련의 숫자, 32비트 컴퓨터에서는 총 $2^{32}$개 등..

⇒ 컴퓨터 상의 주소 = Page, 32비트 중 12비트는 Page를 표현하는 비트로써 사용한다.

1. Logical Address ( Virtual Address )

   ⇒ 프로세스마다 독립적으로 가지는 주소공간, 각 프로세스마다 0번지부터 시작

   ⇒ CPU가 보는 주소는 Logical Address!

2. Physical Address

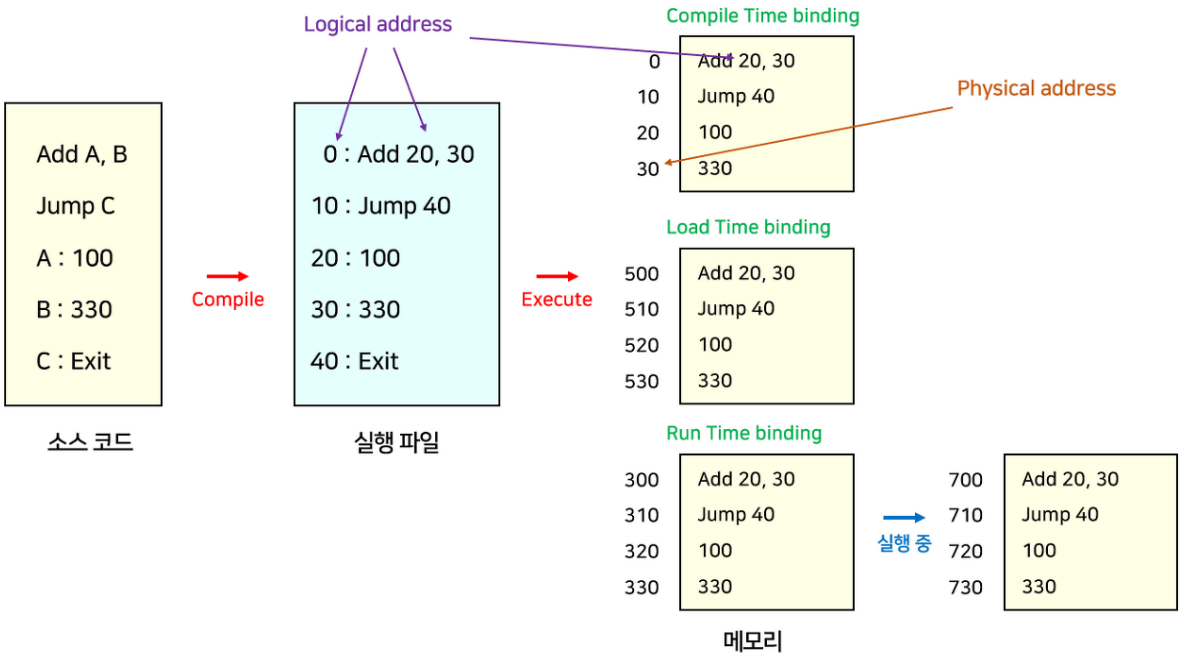
   ⇒ 실제 메모리(RAM)에 올라가는 위치

## 2. 주소 바인딩 ( Address binding )

⇒ 주소를 결정하는 것을 의미, 어떤 프로그램이 메모리의 어느 위치에, 즉 어떤 물리적 주소에 load 될지를 결정하는 과정

- 프로세스가 실행되기 위해선, 프로그램이 물리적 메모리에 적재되어야 함
- CPU가 기계어 명령을 수행하기 위해 논리적 주소를 통해 메모리 참조 시,
- 해당 논리적 주소가 물리적 메모리의 어느 위치에 매핑되는지를 확인해야 함
- 이렇게 논리적 주소를 물리적 메모리 주소로 연결시켜주는 작업을 주소 바인딩

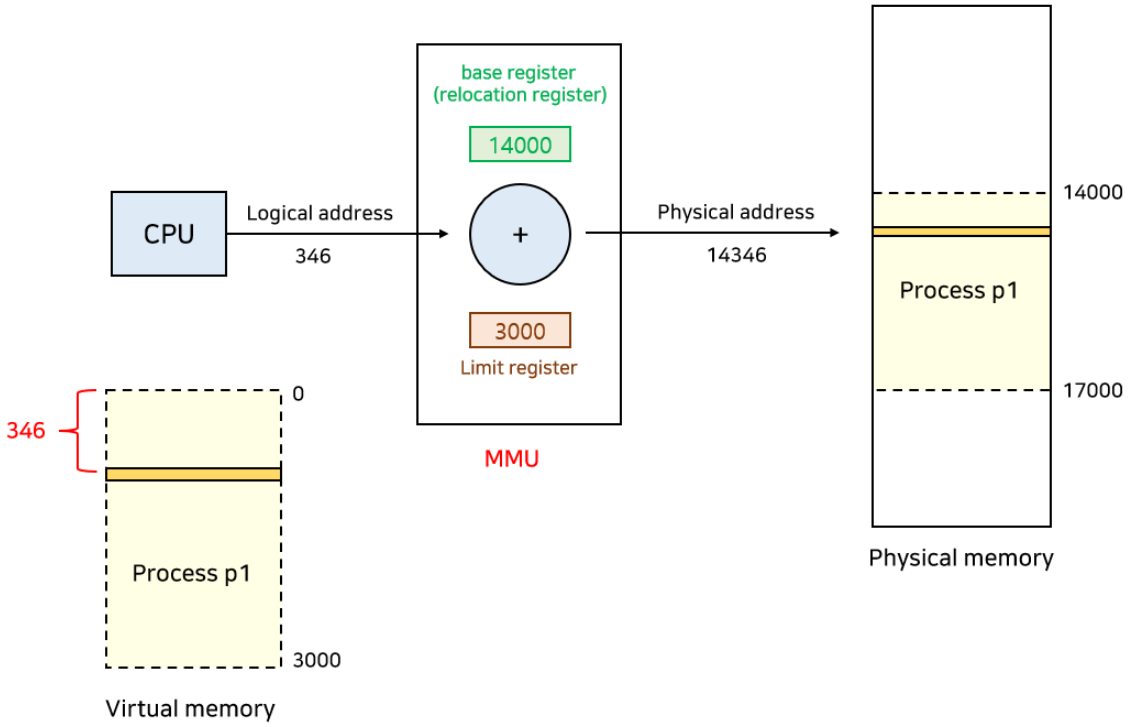
**< 바인딩 시점에 따른 주소 바인딩 분류 >**




1. **Compile time binding**
    - 물리적 메모리 주소가 컴파일 시점에 정해진다.
    - 컴파일러는 “절대코드” 또는 “절대주소” 라는 고정된 주소를 생성 및 사용한다.
    - 시작위치가 변경된다면 재컴파일해야 한다.
    - 컴파일 타임 주소할당은 프로세스 내부에서 사용하는 논리적 주소와 물리적 주소가 동일
    - 현대의 바인딩 방식에선 사용하지 않음

2. **Load time binding**
    - 프로그램이 실행될 때 물리적 메모리 주소가 결정되는 방식
    - Loader의 책임 하에 물리적 메모리 주소가 부여, 프로그램이 종료될 때까지 물리적 메모리  상의 위치가 고정
    - Loader : 사용자 프로그램을 메모리에 적재시키는 프로그램
    - 컴파일러가 “재배치 가능 코드”를 생성한 경우에 가능한 주소 바인딩 방식이다.
    - 컴파일 타임과 마찬가지로 현대 바인딩 방식에 사용되지 않는다.

3. **Execution time binding ( Runtime binding )**




- 프로그램이 실행한 이후에도 그 프로그램이 위치한 물리적 메모리 상의 주소가 변경 가능한  바인딩 방식
- CPU가 주소를 참조할 때 마다, 해당 데이터가 물리적 메모리의 어느 위치에 존재하는 지 address mapping table을 사용해 바인딩을 점검한다.
- 런타임 주소 할당은 MMU( Memory Management Unit )라는 하드웨어 장치를 사용하여,  논리적 주소를 물리적 주소로 바꿔준다.
- MMU 기법은 프로세스가 CPU에 수행하면서 생성해내는 모든 주소값에 대해, base register값을 더해주어 물리적 주소를 생성하는 방식 ( base Register는 1개, Process 끼리 공유! )

- CPU의 논리적 주소 상의 346번지의 정보를 요청 할 때, 해당 주소를 실제 메모리 상에서  얻기 위해 주소 변환과정이 필요하다.
- base register & limit register 2개의 레지스터를 활용해 주소변환을 진행하며, Virtual memory( logical memory )가 올라가 있는 위치에 logical memory 상에 존재하는 해당 번지의 위치를 더해주면 값을 얻는다. ( 346 + 14000, 14000 ~ 17000 )

### ※ Limit register

⇒ 논리적 주소의 범위이며, 잘못된 메모리를 참조하지 않도록 막아주는 기능 수행

⇒ base register는 접근할 수 있는 물리적 주소의 최솟값을 나타냄, 커널 모드인 경우에는 MMU가 물리적인 주소로 변환하지 않고 논리적인 주소를 그대로 사용한다.

⇒ Ex) 가상메모리 최대 크기 = 3000 일 때, 해당 Logical memory에 있지 않는 4000번지를 요구한다면, 범위를 벗어난 다른 프로그램 위치를 요구하기 때문에 그 요청을 거절한다.

---

## 3. 물리적 메모리 할당 관리

⇒ 메모리는 일반적으로 OS 상주영역과 사용자 프로세스 영역 2가지로 구분한다.

⇒ OS 상주영역 : 인터럽트 Vector & 낮은 주소영역으로 할당 + 커널

⇒ 사용자 프로세스 영역 : 높은 주소영역으로 할당 + 사용자 프로세스가 적재

**※ 메모리 파편화**

1. **내부 단편화 ( Internal Fragmentation ) :  조각의 빈칸**

   ⇒ 프로그램 크기보다 분할의 크기가 큰 경우

   ⇒ 하나의 분할 내부에서 발생하는 사용되지 않는 메모리 조각

   ⇒ 특정 프로그램에 배정되었지만 사용되지 않는 공간

2. **외부 단편화 ( External Fragmentation ) : 전체 메모리 중 빈칸**

   ⇒ 프로그램 크기보다 분할의 크기가 작은 경우

   ⇒ 아무 프로그램에도 배정되지 않은 빈 곳인데도, 프로그램이 올라갈 수 없는 작은 분할

**※ 압축 ( Compaction )**

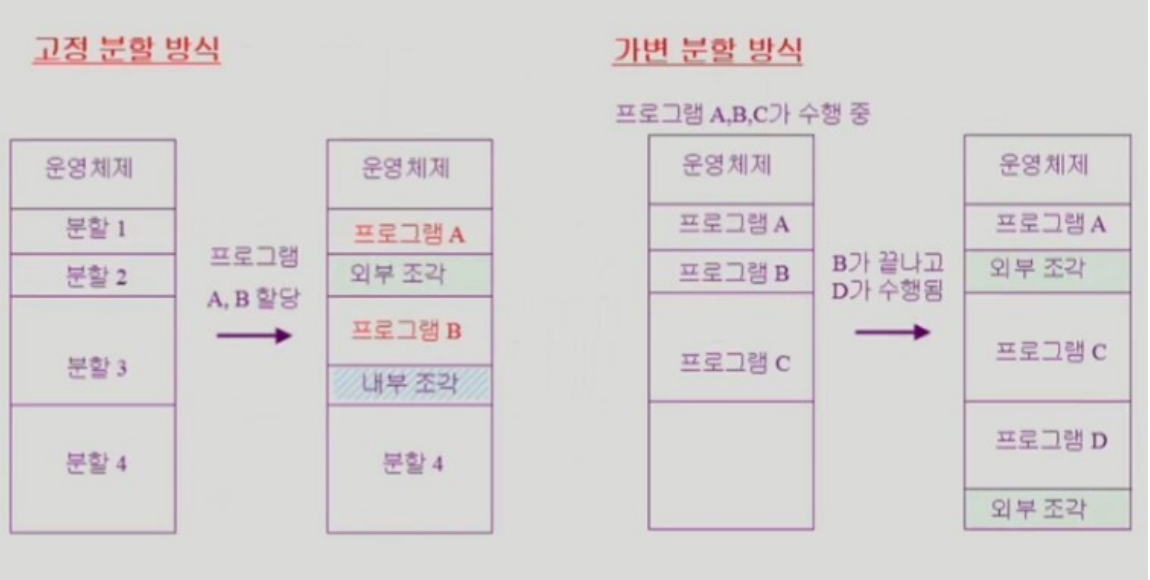
⇒ 외부 단편화를 해소하기 위해 프로세스가 사용하는 공간들을 한쪽으로 몰아, 자유공간을 확보하는 방법론이지만, 작업효율이 좋지 않다.

⇒ process 주소 실행 시간에 동적으로 재배치가 가능한 경우에만 수행될 수 있음

⇒ 한 곳으로 몰고 남은 하나의 큰 가용공간 = block

## < 사용자 프로세스 영역의 메모리 할당 >

## 1. 연속 할당




⇒ 프로세스를 메모리에 올릴 때 그 주소 공간을 여러 개로 분할하지 않고 연속된 메모리 공간을 차지하도록 적재

⇒ 각 프로세스를 메모리에 담기 위해 메모리는 미리 공간을 분할한다,

⇒ 고정된 크기로 나누는 고정 분할 방식과, 프로세스의 크기를 고려해서 나누는 가변 분할 방식

1. **고정 분할 방식**

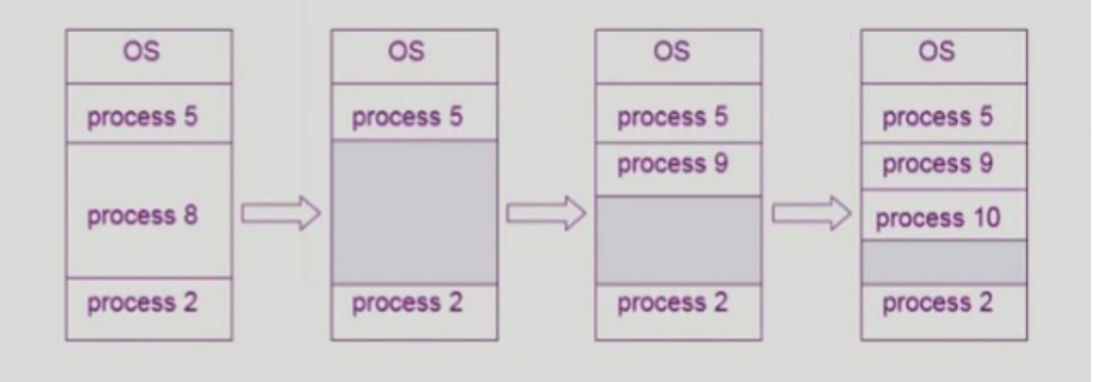
   ⇒ 물리적 메모리를 주어진 개수만큼의 영구적인 분할(partition)으로 나누어 두고 각 분할에 하나의 프로세스를 적재해 실행

   ⇒ 분할 당 하나의 프로세스가 적재 → 동시에 메모리에 load 되는 프로세스의 수가 고정

   ⇒ 실행 가능한 프로세스의 최대 크기가 제한 -> 융통성 ↓

   ⇒ 내부 & 외부 조각이 모두 발생해 낭비가 심함.

2. **가변 분할 방식**




⇒ 프로세스 크기 고려한 할당, 이를 위한 기술적 관리기법 요구, 외부 단편화만 발생

⇒ Contiguous Allocation에서 메모리를 분할하는 각 단위는 Block, 그 중 프로세스가 사용  할 수 있는 Block을 Hole

⇒ 다양한 크기의 Hole들이 메모리 여러 곳에 흩어져 있고, Process가 도착하면 수용 가능한 Hole을 할당

⇒ 할당과 제거를 반복하다보면 Scattered Holes가 생겨나고 이로 인한 외부 단편화가 발생

**< 가변 할당을 위한 기술적 관리 기법, Dynamic Storage-Allocation Problem >**

⇒ 크기가 n인 프로세스가 들어갈 가장 적절한 Hole을 찾는 기법

⇒ 이를 통해, 외부 단편화 현상을 감소시킬 수 있음

**※ First - fit**

⇒ 크기가 n 이상인 Hole 중 최초로 발견한 Hole에 할당, 빠른 장점

**※ Best - fit**

⇒ 크기가 n 이상인 가장 작은 hole, 즉 빈 메모리 공간의 크기와 프로세스의 크기 차이가 가장 적은 곳을 선택

⇒ hole들이 크기순으로 정렬되지 않은 경우 모든 Hole을 탐색해야 한다는 부담

⇒ 항상 거의 딱 맞는 크기를 할당하기 때문에 할당 후에 아주 작은 Hole들이 많이 생성

❄ **Worst - fit**

⇒ 가장 큰 Hole에 할당, 빈 메모리 공간의 크기와 프로세스의 크기 차이가 가장 큰 곳 택

⇒ 모든 Hole을 탐색해야 하고, 상대적으로 아주 큰 Hole들이 새로 생성된다.

⇒ 이렇게 생긴 빈 메모리 공간에 또 다른 프로세스를 할당할 수 있을 거라는 가정에 기인

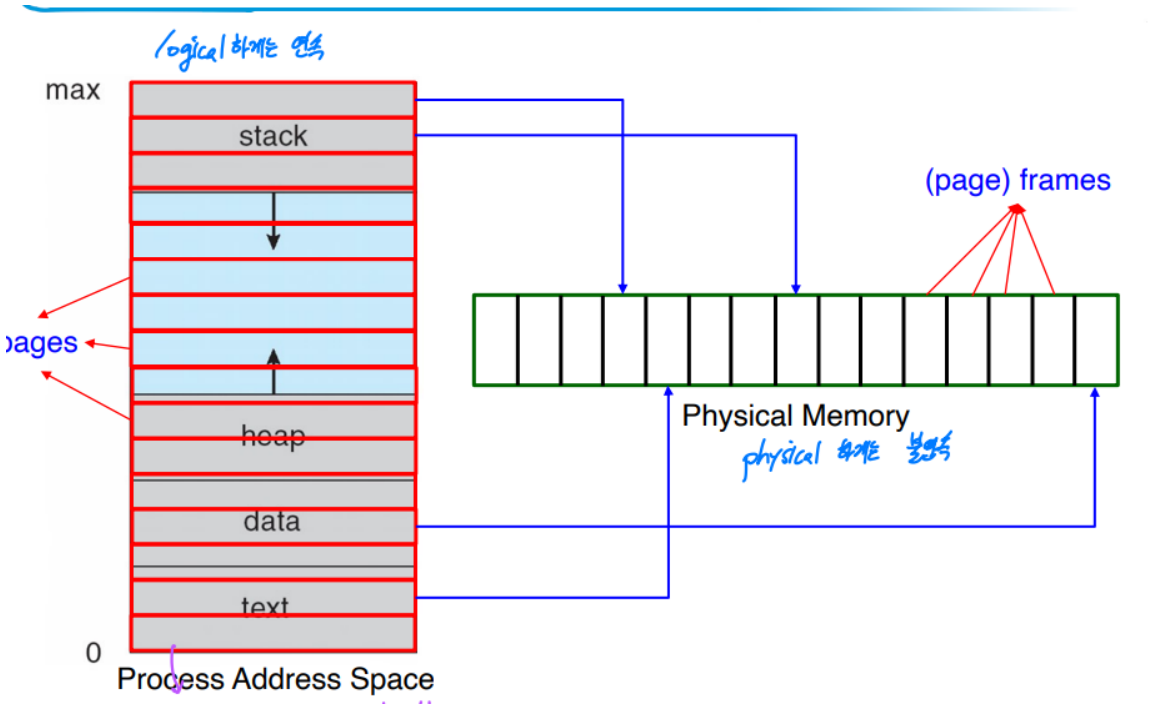
⇒ First-fit과 Best-fit이 Worst-fit에 비해서는 속도나 공간 측면에서 효과적이다.

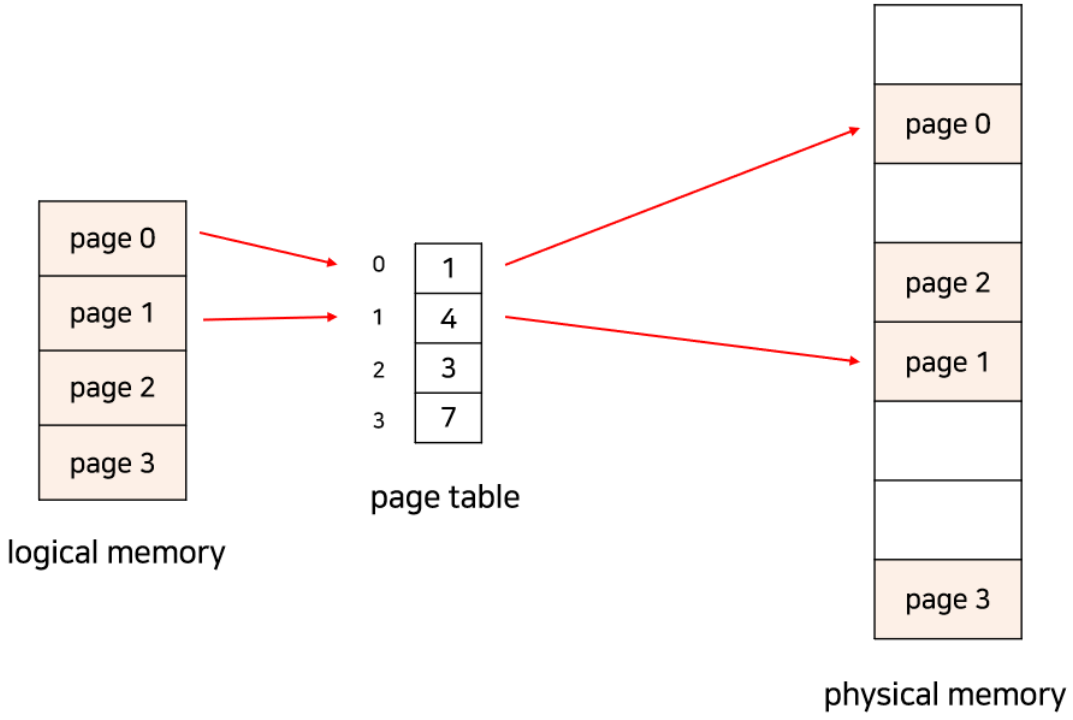
⇒ 하지만 3가지 기법 모두, 전체적으로 효율이 좋지 않은 편이다.

## 2. 불연속 할당 ( Paging )

⇒ 하나의 프로세스가 사용하는 메모리 공간이 연속적이어야 한다는 제약을 없애는 관리 기법

⇒ 외부 단편화 & 압축 비효율성 해소를 위해 탄생한 메모리 관리 기법, 현재 상용화된 기법






1. **페이징 ( Paging )**

   ⇒ 프로세스의 주소 공간을 동일 사이즈의 페이지 단위로 나누어 물리적 메모리에 불연속적으로 저장하는 방식

   ( logical memory = 순차적 & physcial memory = 불연속 )

   ⇒ 물리 메모리 : Frame ( 고정 크기 ) & 논리 메모리 : Page ( 프로세스가 점유하는 고정 크기 )

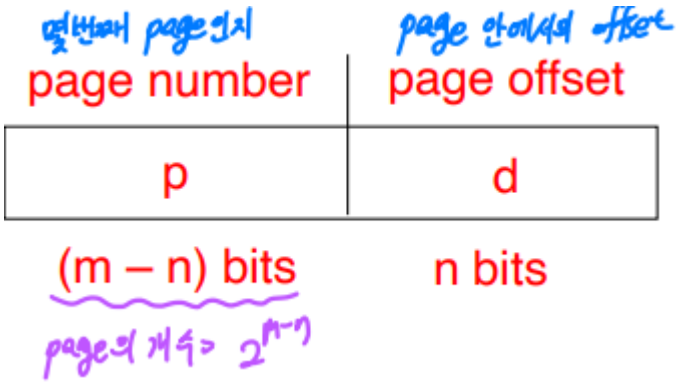
   ⇒ 각 블럭의 크기는 2의 거듭제곱, 한 프로세스의 공간이 여러 page로 나뉘어 관리, 각각의 page 는 순서와 상관없이 frame에 mapping되어 저장 ( 외부 단편화 해결 )

   ⇒ page ↔ frame Mapping에 앞서, 할당 가능한 빈 공간의 frame을 모두 탐색한다.

   ⇒ 프로세스가 순서대로 저장 X → page & frame 연결 관련 정보를 유지하는 자료구조 필요

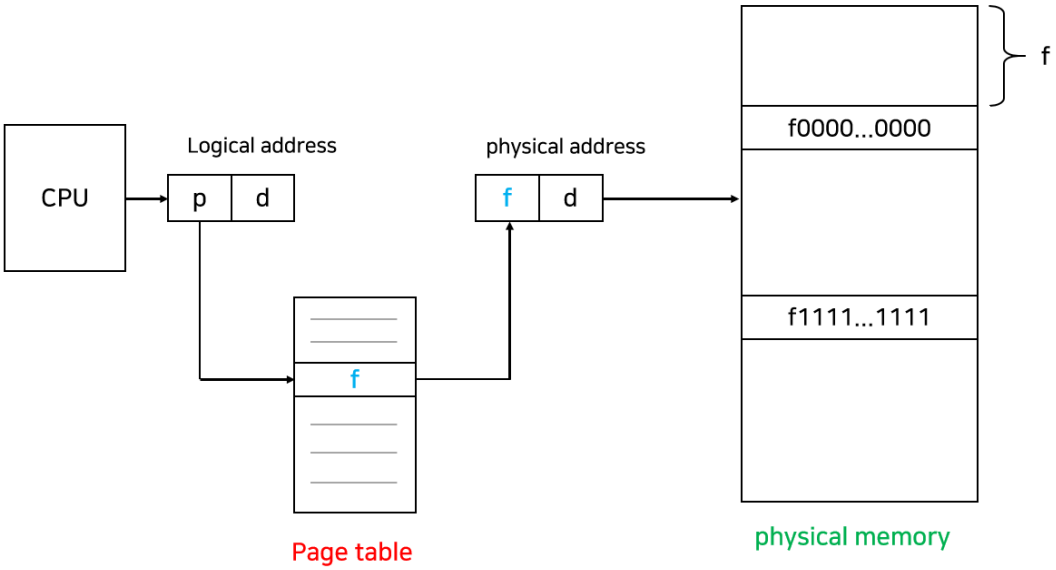
   ⇒ page table : 논리적 주소 ↔ 물리적 주소 매핑하는 해시 테이블 구조

**< 주소 변환 기법 >**




1. **Page Number (p)** : 물리적 메모리에 있는 각 페이지 기본 주소의 Page Table 인덱스
    - 해당 인덱스 항목에는 그 페이지의 물리적 메모리 상의 기본 주소가 저장!
2. **Page Offset (d)** : 기본 주소와 결합하여 나타내는 물리적 메모리 주소범위

( logical address space = $2^m$ && page size = $2^n$ )




⇒ 페이지 테이블의 물리 메모리 기본 주소(P) + 변위 (d) = 해당 page의 물리적 주소

( p = 페이지 번호, d = 페이지 오프셋 )

**❄ Paging 기법의 장점**

⇒ 동적 연속 할당 & 압축 기법의 비효율성 및 문제점을 해결

⇒ page들이 연속하지 않아, 외부 단편화 발생 X

⇒ 메모리 할당과 해제가 빨라 swap out이 간단하다.

⇒ 코드를 쉽게 공유 가능하다.

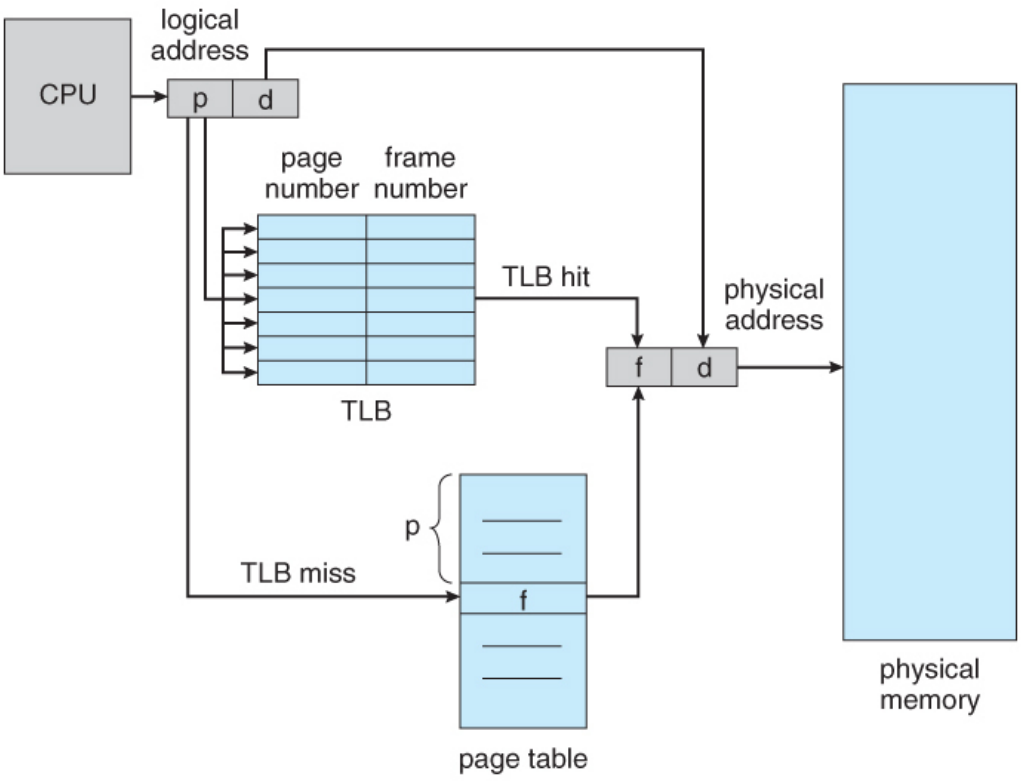
**❄ Paging 기법의 단점**

⇒ 연속할당 기법에 비해 주소변환 절차가 복잡하다.

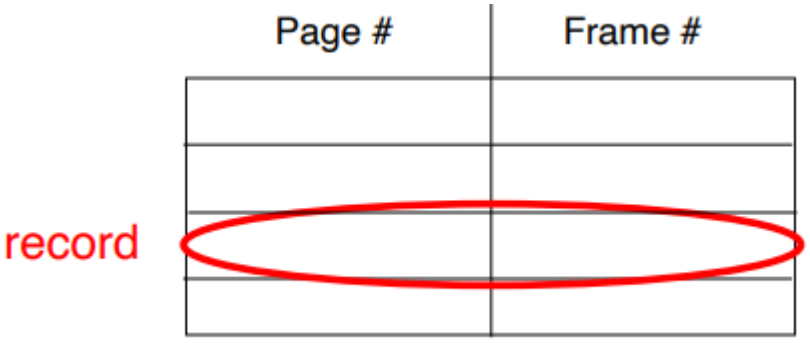
⇒ 내부 단편화를 해결하지 못한다.

⇒ page Table을 저장하기 위한 메모리가 추가 소모, 그리고 테이블이 메모리에 상주하기 때문에, 메모리에 접근하는 연산은 2번( table 접근 + 실제 연산 )
⇒ 속도 ↓

- Page-table base Register (PTBR, table pointer)
- page-table length register (PTLR, page table size)
- 속도 향상을 위해, **Associative register / TLB** 라 불리는 고속 하드웨어 캐시 사용!




**※ TLB ( Associative register로 구성된 병렬 탐색 하드웨어 캐시 )**




⇒ 하드웨어 친화적 캐시, 병렬 탐색으로 2번의 연산을 한 번에 처리! ( Record )

⇒ 데이터 저장 X! 단지 주소변환 연산을 위한 캐시, page table 일부 내용만 가짐

⇒ CPU는 메모리 상 page Table 접근 전, TLB 전체를 탐색한다.

- 주소 변환 정보 중, 캐시에 이미 저장된 것이라면 바로 사용!
- 아니면 page Table을 순차 탐색한다.

⇒ Context Switching 시에, 해당 레지스터 내용만 변경하면 됨 → 페이징 성능 개선!

⇒ Context Switching 발생 시 flush 명령 → 오래된 데이터 제거

⇒ TLB 사용 시 평균 메모리 접근 시간 = EAT ( Effective Mem-Access time )

**※ EAT ( Effective Mem-Access time )**

⇒ CPU가 memory에 접근해서 원하는 data를 가져오는 데 걸리는 평균시간

⇒ ( TLB 탐색 시간 : $a$, Mem 접근 시간 : β, TLB hit ratio( 존재확률, 0 ~ 1 ) : ξ )

⇒ ( TLB 탐색시간보다, Mem 접근 시간이 압도적으로 더 걸린다. )

⇒ EAT : EAT = <hit> + <miss> = (a + β) ξ + (a + 2 β)(1 − ξ) = a + (2 - ξ)

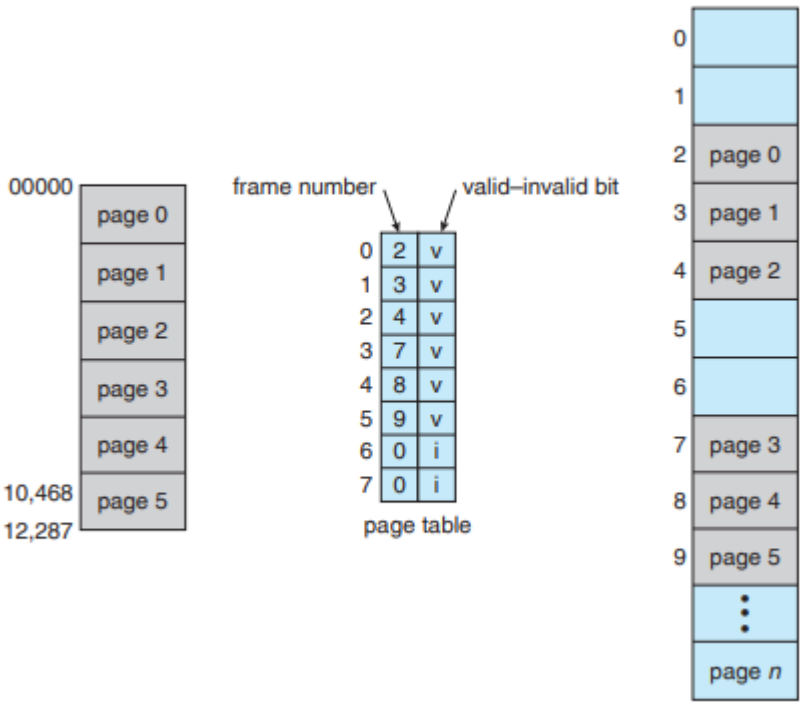
- <miss> : (page table lookup + mem access)  = failed! + mem access
- TLB miss 나면 원하는 record가 없음! 때문에 table에 접근하는 β 소요!
- TLB가 아예 없다면, (a + β) 부분은 필요 없음!

⇒ context switch 발생하면, 초반에 TLB miss가 계속 발생!

**※ 페이지 테이블의 각 Entry에 위치한, 여러 정보를 담는 Bit 종류**

1. **protection bit**
   - page에 대한 접근 권한 ( read / write / read-only )

2. **Valid - Invalid bit**




- Valid : 해당 주소의 frame에 그 프로세스를 구성하는 유효 정보 존재
    - 테이블에서 쓰고있는 entry : 접근 허용
- Invalid : 해당 주소의 frame에 유효한 내용이 없음
    - 테이블에서 쓰고 있지 않는 entry : 접근 불가

⇒ 메모리 할당이 연속하다면, Limit만 비교해도 메모리 보호 가능

⇒ 페이지 기법은 불연속적!

- valid-invalid bit를 사용해 valid인 경우에만 접근 허용!

**※ Valid - Invalid bit 사용 이유?**

⇒ 주소 공간이 변해서, 동적으로 page가 몇 개 필요한지 알기 위해 사용

⇒ 프로세스 내 주소 변조에도 필요! heap이 stack 공간을 침범하는 걸 막는다.

⇒ page 크기 작을수록, 내부 단편화 감소 및 필요 정보만 존재!

- 메모리 이용 효율적

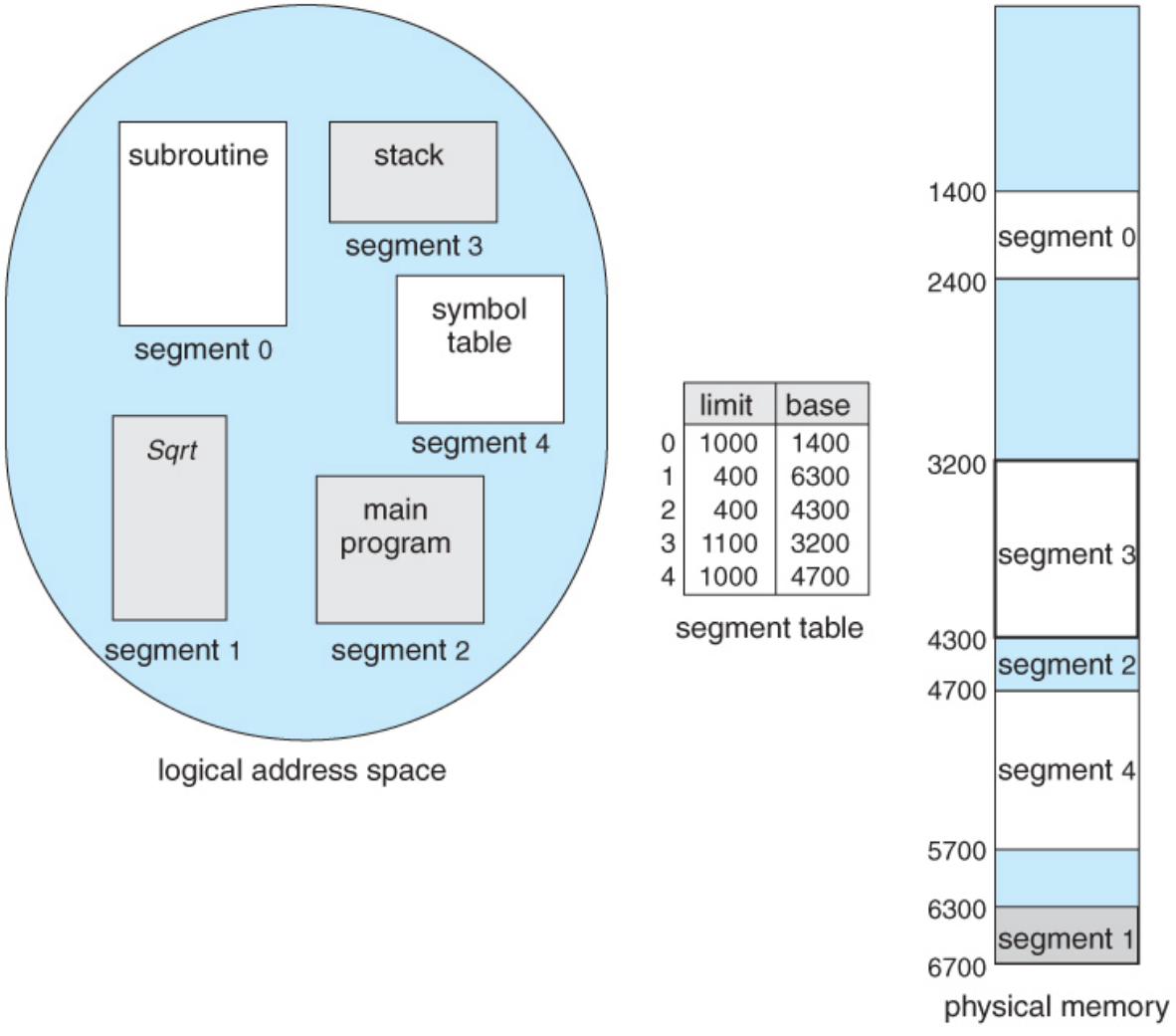
- page table 크기 증가, 디스크 이동 효율성 감소

※ 현대의 운영체제는 page 크기를 키우는 게 트렌드..

※ ( 심화 ) 계층적 페이지

---

## 2. 세그멘테이션 ( Segmentation )




⇒ Segmentation은 의미 단위로 하나의 프로세스를 여러 개의 segment로 나누는 것을 의미

⇒ 각 Segment는 연속하지 않으며, Segment 자체에 포함된 메모리 주소는 연속적이다.

⇒ 작게는 프로그램을 구성하는 각 함수를, 크게는 프로그램 전체를 하나의 Segment로 정의

- Logical unit : main(), function, global variables, stack, symbol table, array 등..

⇒ 일반적으로는 code, data, stack 부분이 하나의 세그먼트로 정의된다.

⇒ Segmentation의 논리적 주소는 segment number, offset으로 구성

⇒ 각각의 segment는 base, limit, protection bit을 가지고 있다.

- ( base, limit ) ⇒ 물리적 주소로 매핑되는 enrty로 구성
- **base** : 메모리에 상주 중인 세그먼트의 물리적 주소의 시작 주소 변수
- **limit** : 세그먼트의 크기, 길이를 명시하는 변수

**※ Segment Table**

⇒ 세그먼트들의 물리 메모리 정보를 저장하는 테이블, 테이블은 메모리에 올라와 있음

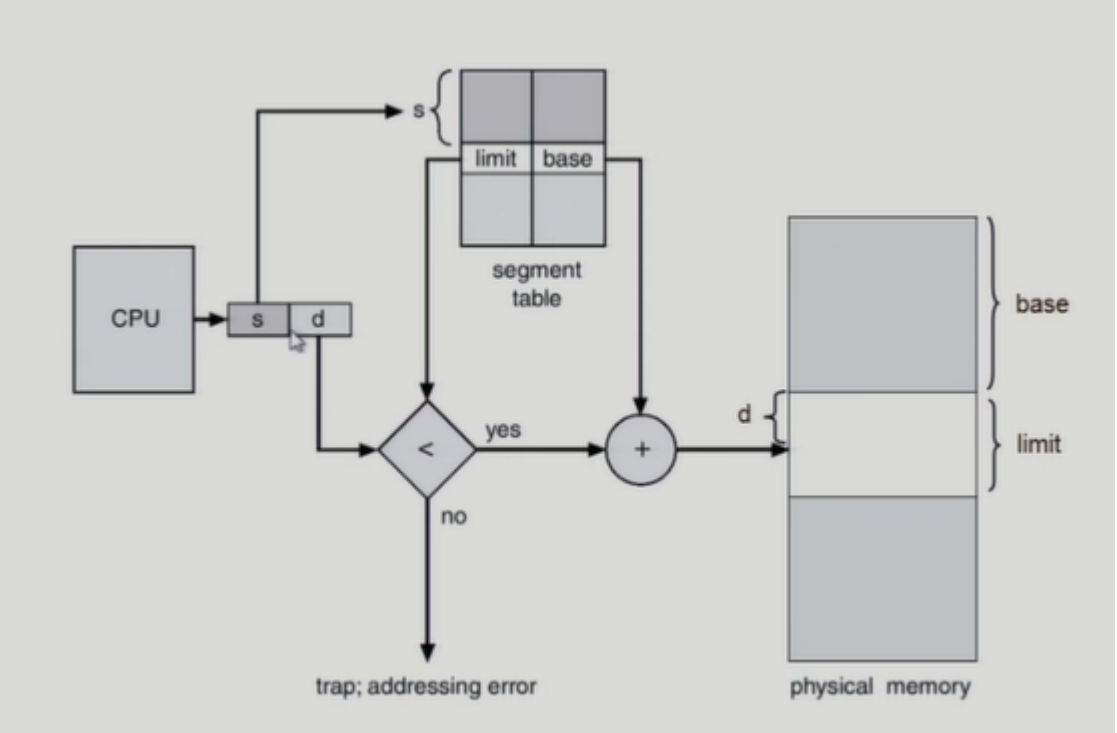
⇒ 해당 세그먼트의 시작 물리 주소 base와 세그먼트 용량을 나타내는 limit의 정보를 담는다.

⇒ 읽기, 쓰기, 실행 등의 권한을 담는 privilege 정보도 담는다.

- Segment-table base register(STBR) : 물리적 메모리에서의 테이블 주소 정보 담은 reg
- Segment-table length register(STLR) : 사용 중인 세그먼트의 개수 정보 ( 테이블 길이 ) 저장

⇒ 이를 통해, 특정 세그먼트 번호 s는 다음과 같은 조건을 만족한다. ( s < STLR )

**< 주소 변환 기법 >**




⇒ 세그멘테이션 기법도 세그먼트 테이블의 각 항목에 보호, 유효 비트가 존재한다.

- Protection bit : 읽기 / 쓰기 / 실행 등의 권한
- Valid - Invalid bit : 주소 변환 정보 유효성 체크

( s : segment num, seg table base reg가 s를 가리키는 중 & d : offset )

1. CPU가 논리 주소를 넘기면 세그먼트 번호(s)와 세그먼트 오프셋(d)으로 나눈다.
2. 세그먼트 시작 위치는 STBR이 가지고 있고, 이를 참조해 세그먼트 번호 s 만큼의 위치로 이동
    - 이를 통해, base와 limit을 얻을 수 있다.
3. 유효성 검증을 한다. STLR에 담긴 세그먼트의 개수 정보보다 s가 작아야 한다.
    - limit의 범위를 넘어가지 않는지 체크를 수행한다.
    - 만약 s가 더 크다면 잘못된 메모리 참조 시도이므로 중단한다. ( Trap )
4. 유효성 검증을 끝냈다면, base와 d를 더한 위치로 이동해 정보를 얻는다.

**※ Sharing Segment ( Segment 단위의 공유 )**


Ex ) Editor 부분을 공유하는 Segment의 예시

⇒ Segement와 연관된 보호 비트 : 이를 통해, Segment 수준에서 Code 공유가 발생

⇒ 공유 세그먼트를 적용하기 위해, 각 프로세스는 같은 세그먼트에 같은 번호를 가져야 한다.

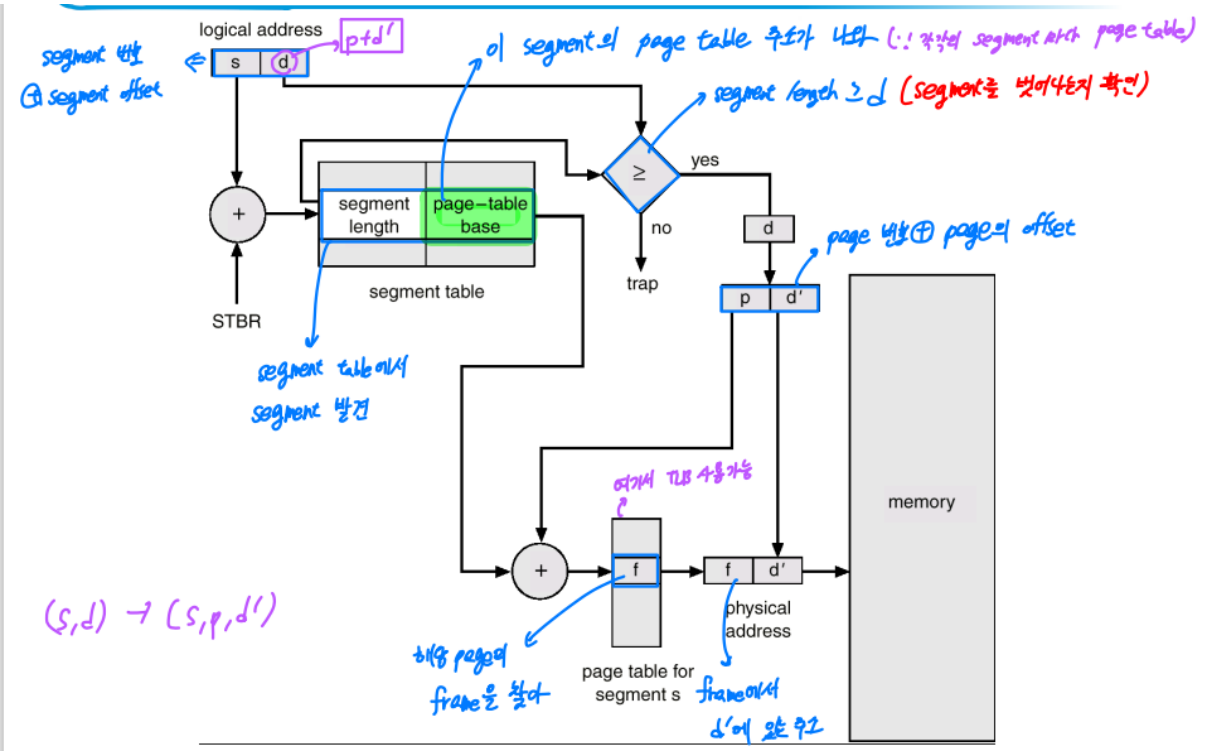
**※ Segmentation 기법의 장점**

- segment들이 연속으로 할당 X, stack과 heap이 독립적으로 커질 수 있다.
- segment마다 protection을 따로 수행 ( paging가 유사 )
- segment는 의미 단위, 공유와 보안에 있어 paging 기법보다 훨씬 효과적

**※ Segmentation 기법의 단점**

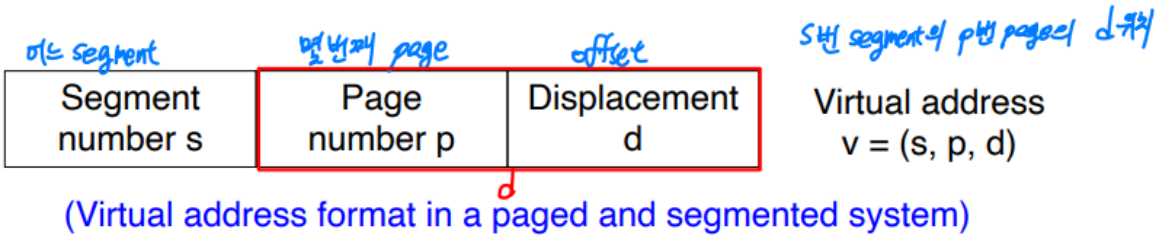
- 각 segment 길이는 가변적이라, 가변 분할 방식에서 동일한 문제점
- 외부 단편화 ( hole ) 발생하나, 내부 단편화는 발생하지 않는다. ( Paging과 반대! )

## ※ Segmentation with Paging




⇒ 페이징 기법과 세그멘테이션 기법의 장점만을 취한 형태

- 세그먼트 페이징 → 외부 단편화도 해결
- segment 단위로 프로세스 간 공유 및 프로세스 내 접근 권한 보호 → paging 약점 해소
- Logical = segment 단위 & physical = Frame 단위 ⇒ 각각의 segment를 page로 처리




⇒ 프로그램을 의미 단위로 나누되, segment가 반드시 동일 길이의 page 집합으로 구성

⇒ segment 마다 page table 주소가 필요 → page table 주소를 base로 활용!

※ 기존 base는 더이상 X! → segment 내부는 불연속적으로 바뀔 수 있다.

**< 주소 변환 기법 >**

1. 논리적 주소의 상위 비트인 세그먼트의 번호를 통해 세그먼트 테이블에 접근한다.
2. 세그먼트의 길이와 페이지 테이블 시작 주소를 획득한다.
3. 세그먼트의 길이값과 오프셋값을 비교하는 유효성 검증을 한다.
    - 오프셋이 더 크다면 유효하지 않으므로 트랩을 발생시킨다.
4. 페이지 테이블의 시작 위치 get, 그 위치에서 페이지 번호만큼 떨어진 페이지 테이블 항목 go
5. 해당 테이블 항목에서 물리적 메모리의 페이지 프레임 위치를 얻을 수 있다.

---