8. Memory Management

속성

1. Logical vs. Physical Address

- 메모리 : 주소를 통해 접근하는 매체, 메모리에는 주소가 매겨진다.

1. Logical address(=virtual address; 논리적)

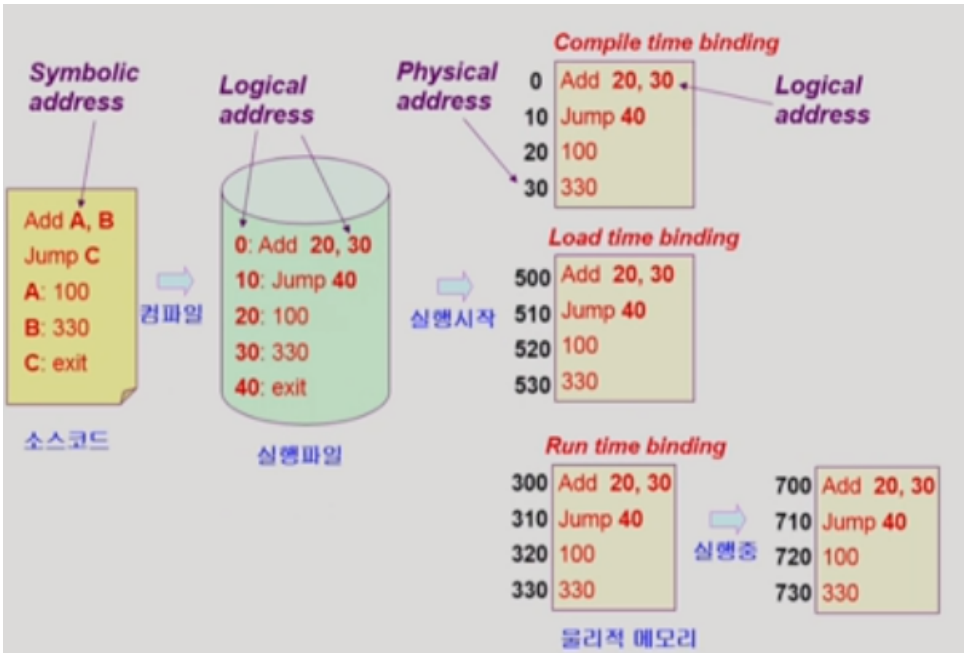
- 프로세스마다 독립적으로 가지는 주소 공간
- 각 프로세스마다 0번지부터 시작
- CPU가 보는 주소는 logical address임

2. Physical address(물리적)

- 메모리에 실제 올라가는 위치
- 프로세스가 실행 되기 위해서는 Logical address에서 Physical address로 올라가야 한다.

- 주소 바인딩 : 주소를 결정하는 것, 물리적인 주소를 갖게 되는 것
  - Symbolic Address ➡️ Logical Address ➡️ Physical address
  - Symbolic Address : 프로그래머들은 몇 번 주소에서 데이터를 가져오는 것이 아니라 변수명을 통해 데이터를 가져온다.
- 주소는 언제 결정 되는냐? (Logical Address에서 Physical address로 바뀌는 시점이 언제 인가?)

2. 주소바인딩(Address Binding)



➡️ Symbolic address에서 Logical address 로 넘어가게 되면 숫자로 이루어진 주소를 가지게 된다.

1. Compile time binding

- 물리적 메모리 주소가 컴파일 시 알려짐
- 시작 위치 변경시 재컴파일
- 컴파일러는 절대 코드(absolute code)생성
  - 만약 주소를 바꾸고 싶다면 컴파일을 다시 해야함.

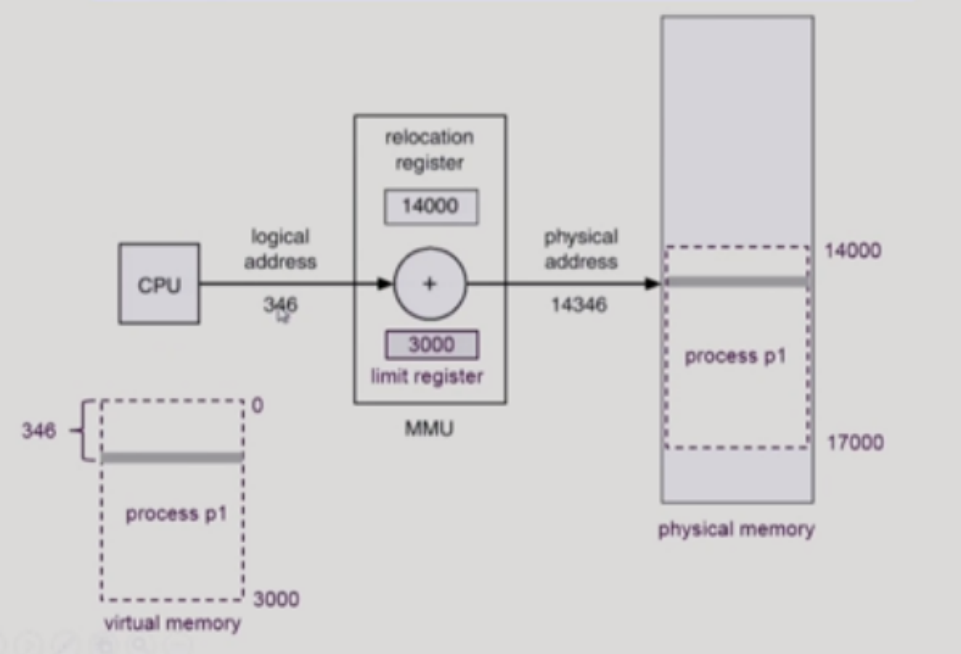
2. Load time binding

- Loader의 책임하에 실행 시작되면 물리적 메모리 주소 부여
- 컴파일러가 재배치가능코드(relocatable code)를 생성한 경우 가능

3. Execution time binding (=Run time binding)

- 수행이 시작된 이후에도 프로세스의 메모리 상 위치를 옮길 수 있음.
- CPU가 주소를 참조할 대마다 binding을 점검(address mapping table)
- 주소 변환시 하드웨어적인 지원이 필요 (ex. base and limit registers, **MMU**)
  - Logical address를 physical address로 매핑해 주는 Hardware device이다.
  - MMU scheme : 사용자 프로세스가 CPU에서 수행되며 생성해내는 모든 주소값에 대해 base register(=relocation register)의 값을 더 한다.

- user program : logical address만을 다룬다. 실제 physical address를 볼 수 없으며 알 필요가 없다.

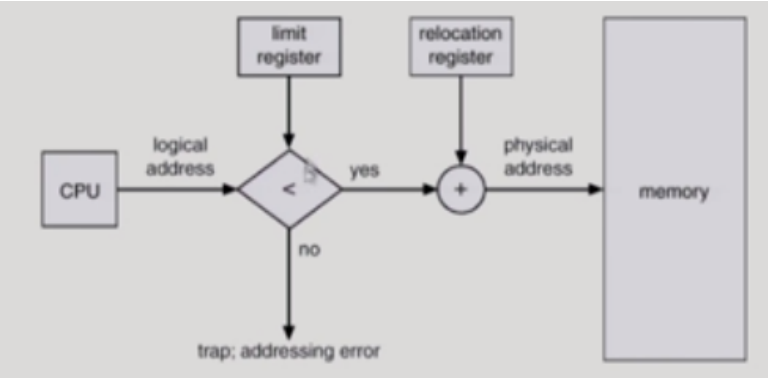


➡️ CPU가 메모리 346번지의 내용을 달라는 것은 logical address이다. 이 경우 주소 변환이 필요하고, MMU로 주소 변화를 한다.

[Dynamic Relocation]

- relocation register(=base register) : 접근할 수 있는 물리적 메모리 주소의 최소값
- limit register : 프로그램의 최대 크기(P1은 3000)를 담고 있다. 악의적인 요청을 막을 수 있다. (P1은 3000까지 있는데 4000번의 주소를 달라고 하는 경우 → 남의 프로그램을 침범 할 수 있다.)

[Hadware Support for Address Translation]



- limit register의 값과 요청된 logical address의 값을 비교한다. 값을 벗어나는 요청이면 trap이 걸리고 CPU제어권이 운영체제에 넘어가고 trap이 걸린 이유를 찾아냄.

3. Some Treminologies

1. Dynamic Loading

- 프로세스 전체를 메모리에 미리 다 올리는 것이 아니라 해당 루틴이 불려질 때 메모리에 load하는 것
- memory utilization의 향상
- 가끔씩 사용되는 많은 양의 코드의 경우 유용(오류 처리 루틴)
- 운영체제의 특별한 지원 없이 프로그램 자체에서 구현 가능(OS는 라이브러리를 통해 지원 가능)
- Loading : 메모리로 올리는 것

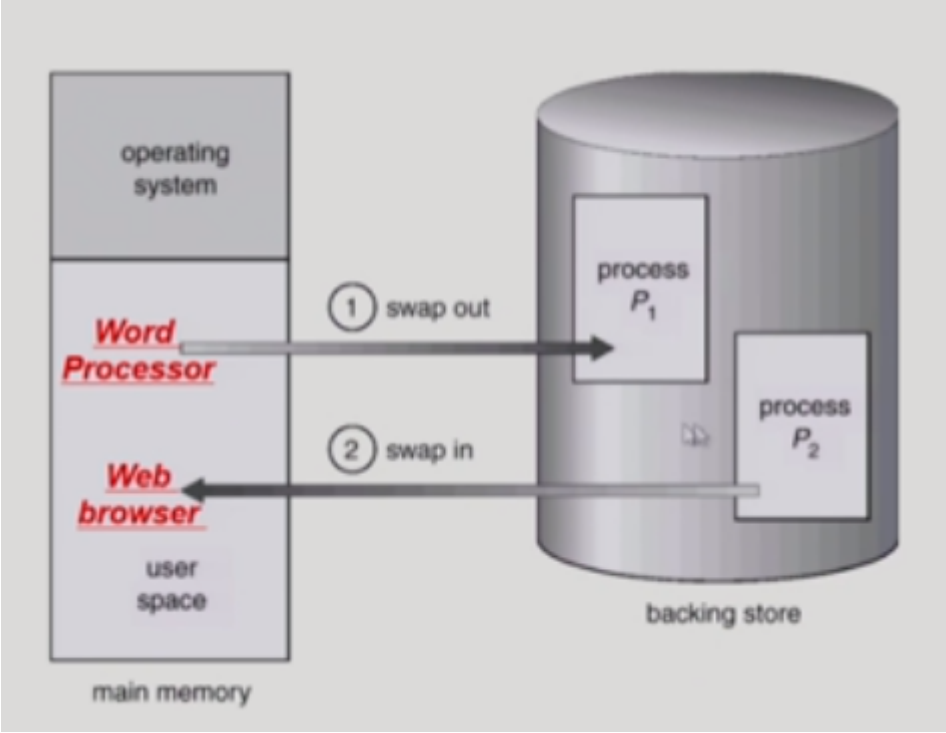
2. Overlays

- 메모리에 프로세스의 부분 중 실제 필요한 정보만을 올림
- 프로세스의 크기가 메모리보다 클 때 유용
- 운영체제의 지원없이 사용자에 의해 구현
- 작은 공간의 메모리를 사용하던 초창기 시스템에서 수작업으로 프로그래머가 구현 (Dynamic Loading과 비슷하지만 다름)
  - Manual Overlay
  - 프로그래밍이 매우 복잡

3. Swapping

- 프로세스를 일시적으로 메모리에서 backing store로 쫓아내는 것
- Backing store(=swap area)
  - 디스크 : 많은 사용자의 프로세스 이미지를 담을 만큼 충분히 빠르고 큰 저장 공간

- Swap in / Swap out
  - 일반적으로 중기 스케줄러(swapper)에 의해 swap out 시킬 프로세스 선정
  - Priority-based CPU scheduling algorithm
    - 우선순위가 낮은 프로세스를 swapped out 시킴
    - 우선순위가 높은 프로세스를 메모리에 올려 놓음
  - Compile time 혹은 load time binding에서는 원래 메모리 위치로 swap in 해야 함
  - Execution time binding(=Run time binding)에서는 추후 빈 메모리 영역 아무 곳에나 올릴 수 있음
  - swap time은 대부분 transfer time(swap되는 양에 비례하는 시간)임



4. Dynamic Linking

- Linking을 실행 시간(execution time)까지 미루는 기법
- Linking이란 여러 군데에서 컴파일 된 파일을 하나로 만드는 것이다.
- Static linking

- 라이브러리가 프로그램의 실행 파일 코드에 포함 됨
- 실행 파일의 크기가 커짐
- 동일한 라이브러리를 각각의 프로세스가 메모리에 올리므로 메모리 낭비 (ex. printf 함수의 라이브러리 코드)

- **Dynamic Linking**
  - 라이브러리가 실행 시 연결(link)됨
  - 라이브러리 호출 부분에 라이브러리 루틴의 위치를 찾기 위한 stub이라는 작은 코드를 둠
  - 라이브러리가 이미 메모리에 있으면 그 루틴의 주소가 가고 없으면 디스크에서 읽어옴
  - 운영체제의 도움이 필요

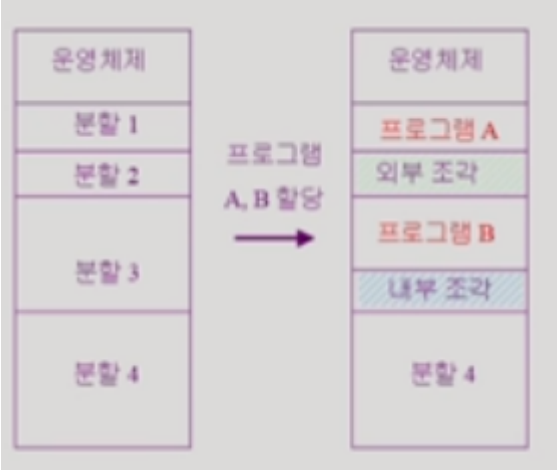
4. Allocation of Physical Memory

- 물리적인 메모리를 어떻게 관리할 것인가?
- 메모리는 일반적으로 두 영역으로 나뉘어 사용
  - OS 상주 영역 : interrupt vector와 함께 낮은 주소 영역 사용
  - 사용자 프로세스 영역 : 높은 주소 영역 사용
- 사용자 프로세스 영역의 할당 방법
  1. **Contiguous allocation(연속할당)**
     a. 각각의 프로세스가 메모리의 연속적인 공간에 적재되도록 하는 것
     b. Fixed partition allocation
     c. Variable partition allocation

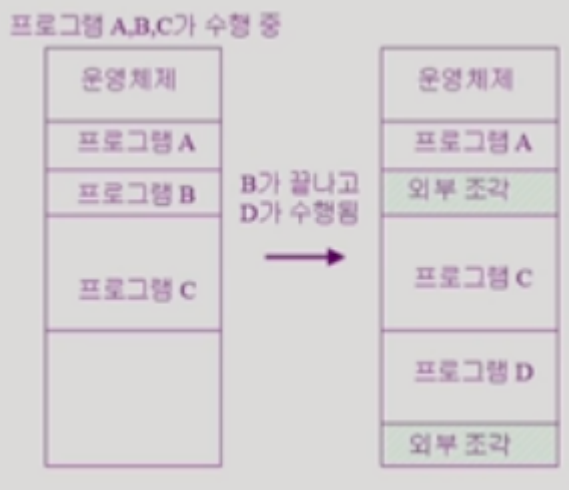
2. **Noncontiguous allocation(불연속 할당)**
    a. 하나의 프로세스가 메모리의 여러 영역에 분산되어 올라갈 수 있음
    b. Paging
    c. Segmentation
    d. Paged Segmentation

1. Contiguous Allocation(연속 할당)

1. 고정 분할 방식(Fixed partition allocation)
    - 낭비되는 조각 발생
    - 외부 조각
        - 올리려는 프로그램 크기보다 메모리 크기가 작아서 사용되지 못한 분할 2
    - 내부 조각
        - 올리려는 프로그램 크기보다 메모리 크기가 커서 안에서 남는 메모리 조각



2. 가변 분할 방식(Variable partition allocation)
    - 프로그램이 실행될 때마다 차곡차곡 메모리에 올려놓는 방법
    - 프로그램 B가 끝나면 프로그램 D가 수행되어야하는데 프로그램 D는 프로그램 B보다 크기가 커서 프로그램 C 다음에 할당된다.
    - 중간 중간 외부조각이 생기게 된다.



- Hole
  - 가용 메모리 공간
  - 다양한 크기의 hole들이 메모리 여러 곳에 흩어져 있다.
  - 프로세스가 도착하면 수용가능한 hole을 할당
  - 운영체제는 다음의 정보를 유지한다.
    - 할당 공간
    - 가용 공간(hole)
  - 가변 분할 방식에서 size n인 요청을 만족하는 가장 적절한 hole을 찾는 문제 (**Dynamic Storage-Allocation Problem**)
    - First-fit : size가 n이상인 것 중 최초로 찾아지는 hole에 할당
    - Best-fit
      - size가 n이상인 가장 작은 hole을 찾아서 할당
      - hole들의 리스트가 크기 순으로 정렬되지 않은 경우 모든 hole의 리스트를 탐색해야 한다.
      - 많은 수의 아주 작은 hole들이 생성된다.
    - Worst-fit
      - 가장 큰 hole에 할당
      - 역시 모든 리스트를 탐색해야 함
      - 상대적으로 아주 큰 hole들이 생성됨

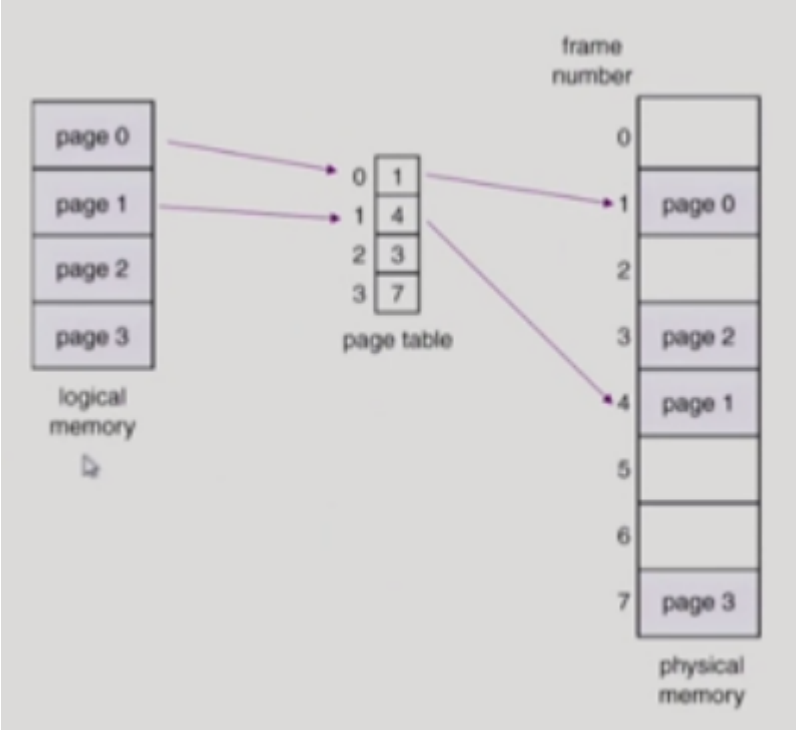
❗ First-fit과 best-fit이 worst-fit보다 속도와 공간 이용률 측면에서 효과적인 것으로 알려짐(실험적 결과)

- Compaction
  - external fragmentation 문제를 해결하는 한가지 방법
  - 사용중인 메모리 영역을 한군데로 몰고 hole들을 다른 한 곳으로 몰아 큰 block을 만드는 것
  - 매우 비용이 많이 드는 방법임
  - 최소한의 메모리 이동으로 compaction하는 방법(매우 복잡한 문제)
  - compaction은 프로세스의 주소가 실행 시간에 동적으로 재배치 가능한 경우에만 수행될 수 있다.

2. Paging(불연속 할당)

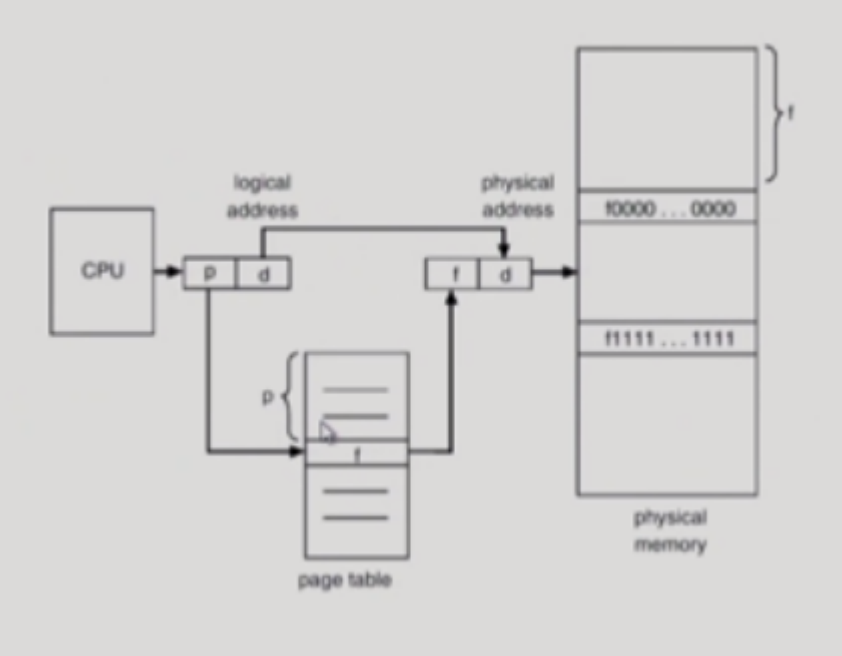
1. **Paging**

- Process의 virtual memory를 동일한 사이즈의 page 단위로 나눔
- Virtual memory의 내용이 page 단위로 noncontiguous하게 저장됨
- 일부는 backing storage에, 일부는 physical memory에 저장
- 주소 변환이 간단하지 않음



n 번째 페이지를 주소 변환하고 싶으면 page table에서 n번째 엔트리를 찾으면 된다.

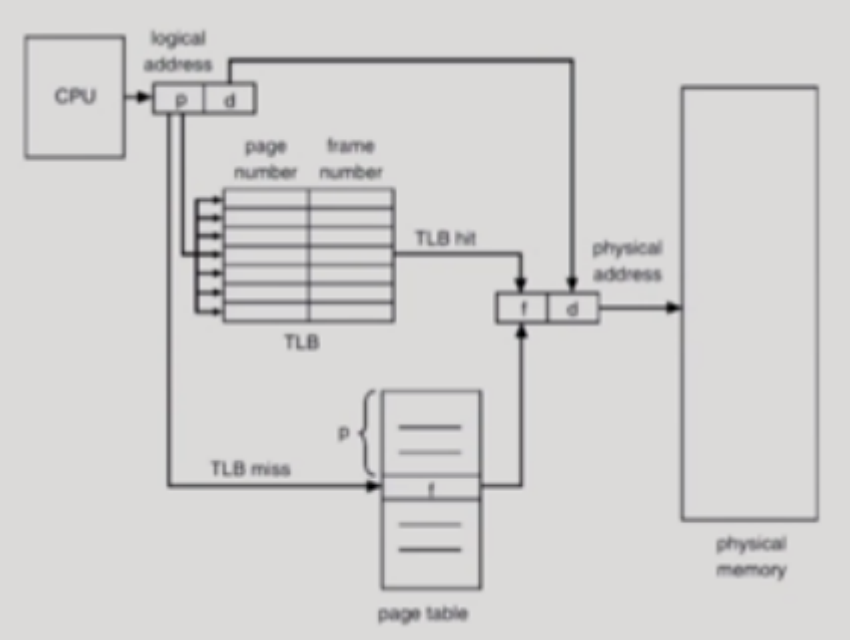
- **Basic Method**
  - physical memeory를 동일한 크기의 frame으로 나눔
  - logical memory를 동일한 크기의 page로 나눔(frame과 같은 크기)
  - 모든 가용 frame들을 관리
  - page table을 사용하여 logical address를 physical address로 변환
  - External fragmentation 발생 안함 : 같은 크기로 잘랐기 때문
  - Internal fragmentation 발생 가능 : 프로그램의 크기가 반드시 page 크기의 배수가 되는 보장이 없기 때문에 마지막에는 page 크기보다 남은 것이 작을 수 있음(내부조각이 생길 수 있다).
- Address Translation Architecture



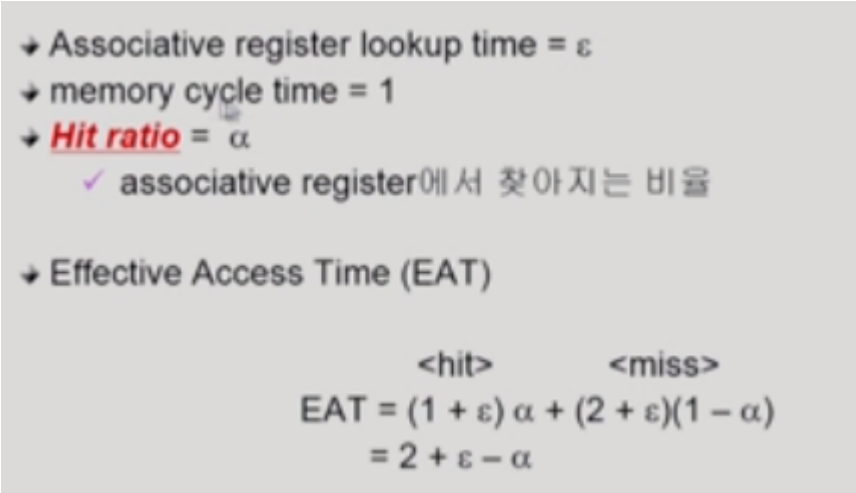
주소에서 앞부분이 페이지 번호가 되고, 뒷부분이 페이지 내에서 얼만큼 떨어져 있는지 나타내는 오프셋이 된다. p가 page table에서 찾아진 값에 의해 변환되어 f로 바뀌게 된다.(물리적 주소로 변경)

- Implementation of Page Table
  - page table은 main memory에 상주한다.
  - page-table base register(PTBR)가 page table을 가리킴
  - page-table lenght register(PTLR)가 테이블 크기를 보관
  - 모든 메모리 접근 연사에는 2번의 memory access 필요
  - *page table 접근 1번, 실제 data/instruction 접근 1번*
  - 속도 향상을 위해 associative register 혹은 translation look-aside buffer(TLB)라 불리는 고속의 lookup hardware cache 사용
  - [Paging Hardware with TLB]
    - main memory 윗단에 cache memory가 있다.
    - 주소 변환을 위한 cache memory라고 할 수 있다.
    - 메인 메모리보다 접근이 빠른 하드웨어이고, CPU가 논리적 주소를 주게 되면 page table에 가기 전에 TLB를 먼저 접근해 TLB에 저장되어 있는 정보를 이용해 주소 변환이 가능한지를 확인한다. 이 경우 한번의 변환이 필요하다.
    - TLB에는 page table 중 일부만 존재한다.

- TLB는 context switch 때마다 flush(remove old entries)된다. 프로세스마다 주소변환 정보다 다르기 때문이다.
- Associative Register
  - CPU가 페이지 정보 P를 주면 P에 대한 정보가 있는지 탐색함
  - parallel search가 가능하다.

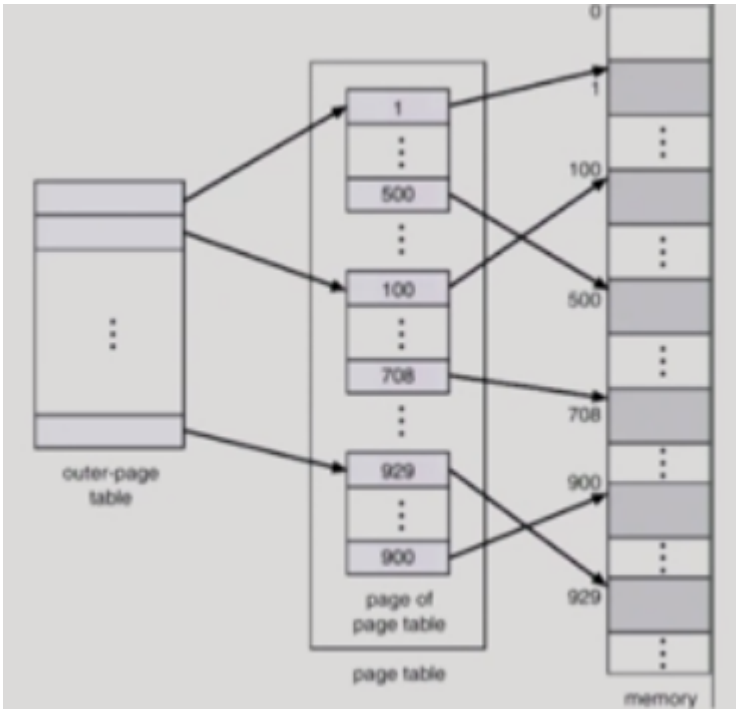


- 실제 메모리 접근하는 시간 (Effective Access Time)

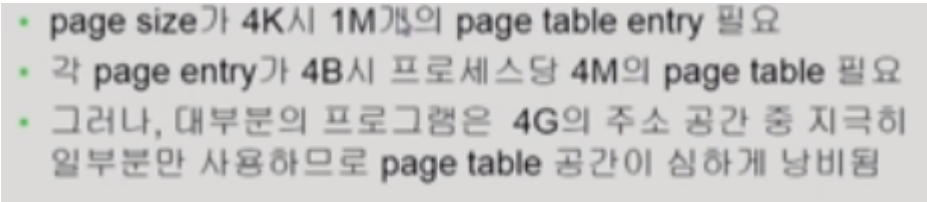


- Two-Level Page Table
  - 현대의 컴퓨터는 address space가 매우 큰 프로그램 지원
  - 두단계의 페이지 테이블을 거침(ex. 행정구역)

- 페이지 테이블 공간이 줄어든다.
- 시간은 더 걸린다.

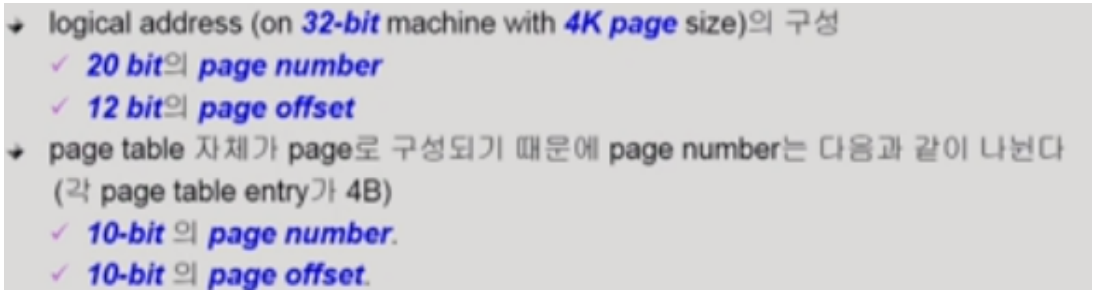


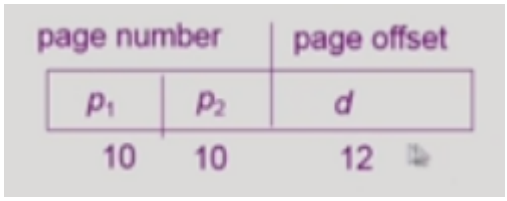
- 32bit 주소 체계를 기준으로 설명 : 2^32 byte(4GB)의 주소공간



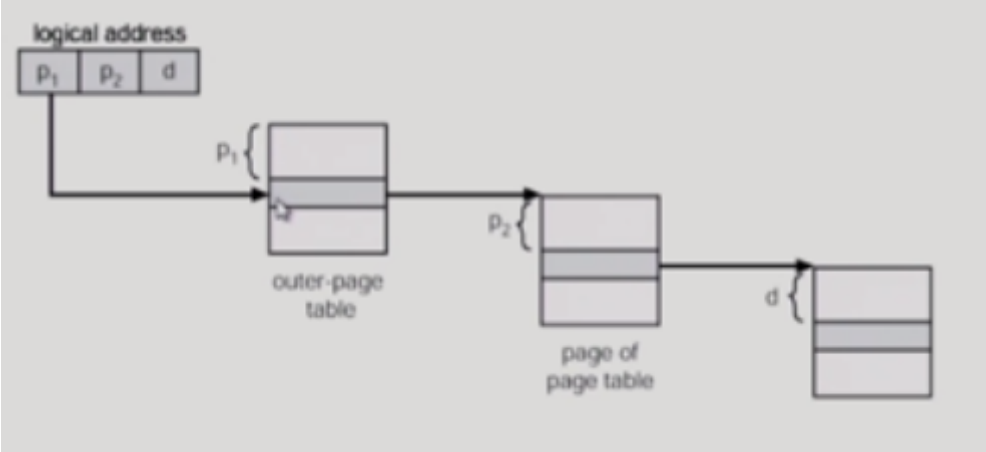
➡️ page table 자체를 page로 구성, 사용이 안되는 페이지가 있다해도 배열이기 때문에 모두 만들어져야 하는데 2단계 페이지 테이블은 이를 해소할 수 있다. 안쪽 페이지 테이블이 안만들어질 수도 있다.

- Two-Level Paging Example





  - P1은 outer page table의 index이고, P2는 outer page table의 page에서의 변위(displacement)이다.
  - 페이지 하나의 크기 4K, 4K byte를 구분하기 위해서 2^12 bit가 필요하다.
  - P2는 1K가 있다. 2^10가지를 구분하기 위해 10bit가 필요하다. P1은 (32-10-12) 10bit가 필요하다.
- Address-Translation Scheme(2단계 주소 변화 과정)

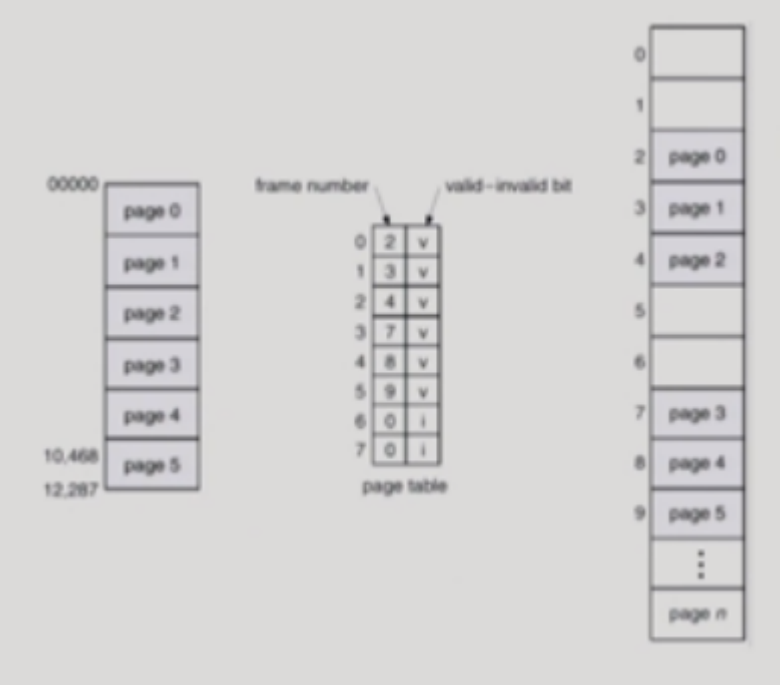


- Multilevel Paging and Performance(다단계)
  - Address space가 더 커지면 다단계 페이지 테이블 필요
  - 각 단계의 페이지 테이블이 메로리에 존재하므로 logical address의 physical address 변환에 더 많은 메모리 접근이 필요
  - TLB를 통해 메모리 접근 시간을 줄일 수 있음
  - 4단계 페이지 테이블을 사용하는 경우
    - 메모리 접근 시간이 100ms, TLB 접근 시간이 20ns이고 TLB hit ratio가 98%인 경우 결과적으로 주소 변환을 위해 28ns만 소요된다.

effective memory access time = 0.98 x 120 + 0.02 x 520
= 128 nanoseconds.

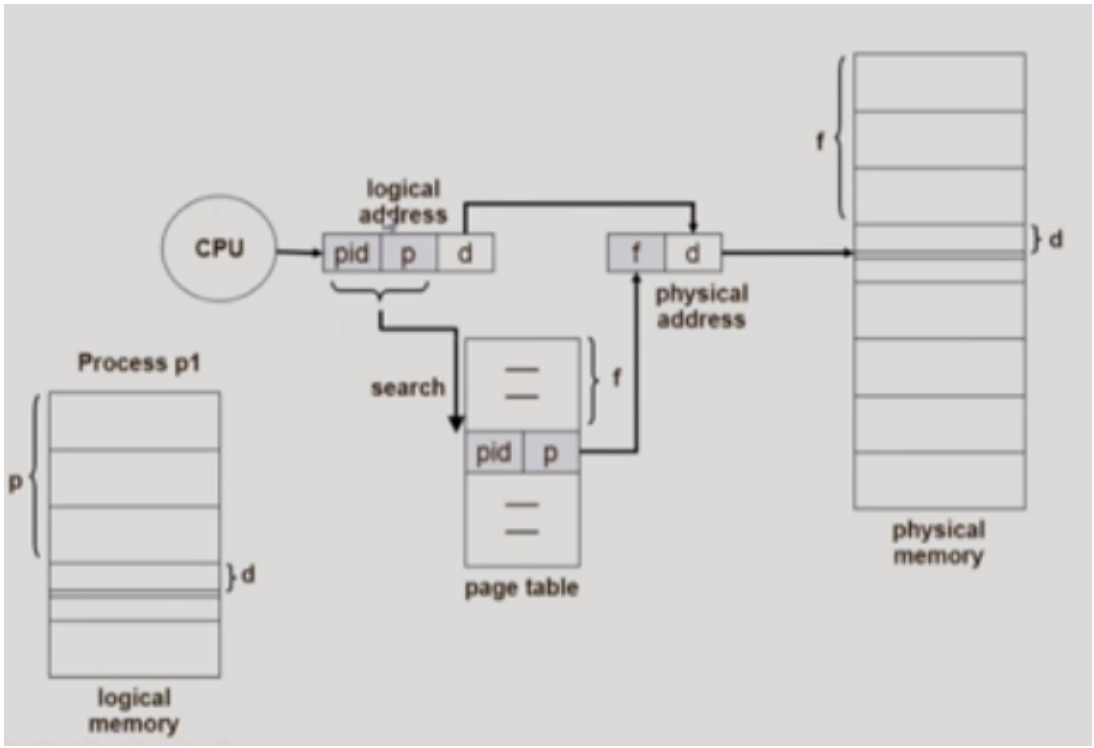
- Valid (v)/ Invalid (i) Bit in a Page Table

- 페이지 테이블에 주소 변환 정보 뿐 아니라 부가적인 정보가 담겨 있다. 6번 7번 엔트리는 사용되지 않지만 page table 엔트리는 생겨야 하기 때문에 만들어져 있고, `i` 라고 쓰여 있기 때문에 사용하지 않는다는 것을 알 수 있다.

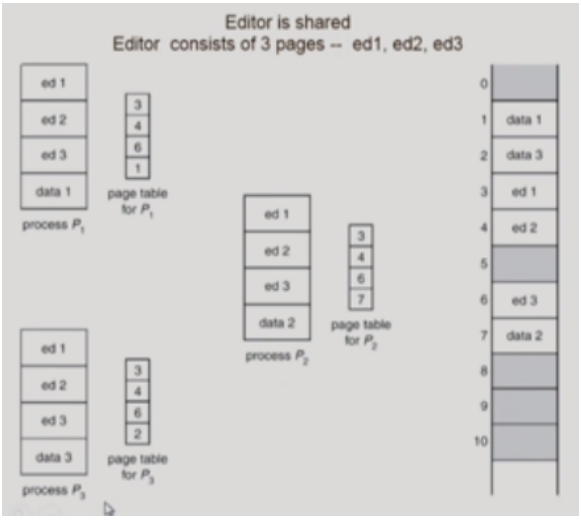


- Memory Protection
  - page table의 각 entry 마다 아래의 bit를 둔다.
    1. Protection bit : page에 대한 접근 권한(read/write/read-only)
    2. Valid-invalid bit : `valid` 는 해당 주소의 frame에 그 프로세스를 구성하는 유효한 내용이 있음을 뜻함(접근 허용). `invalid` 는 해당 주소의 frame에 유효한 내용이 없음을 듯함(접근 불허; swap area에 있는 경우)
- Inverted Page Table
  - page table이 매우 큰 이유
    - 모든 process 별로 그 logical address에 대응하는 모든 page에 대해 page table entry가 존재
    - 대응하는 page가 메모리에 있든 아니든 간에 page table에는 entry로 존재
  - Inverted page table
    - 페이지 테이블 공간을 줄일 수 있지만 시간은 오래 걸림.
    - Page frame 하나당 page table에 하나의 entry를 둔 것이다.(system-wide)

- 각 page table entry는 각각의 물리적 메모리의 page frame이 담고 있는 내용을 표시한다. pid도 포함되어야 한다. (process-id, process의 logical address)
- [단점] : 테이블 전체를 탐색해야 함
- [조치] : associative register 사용(expensive)



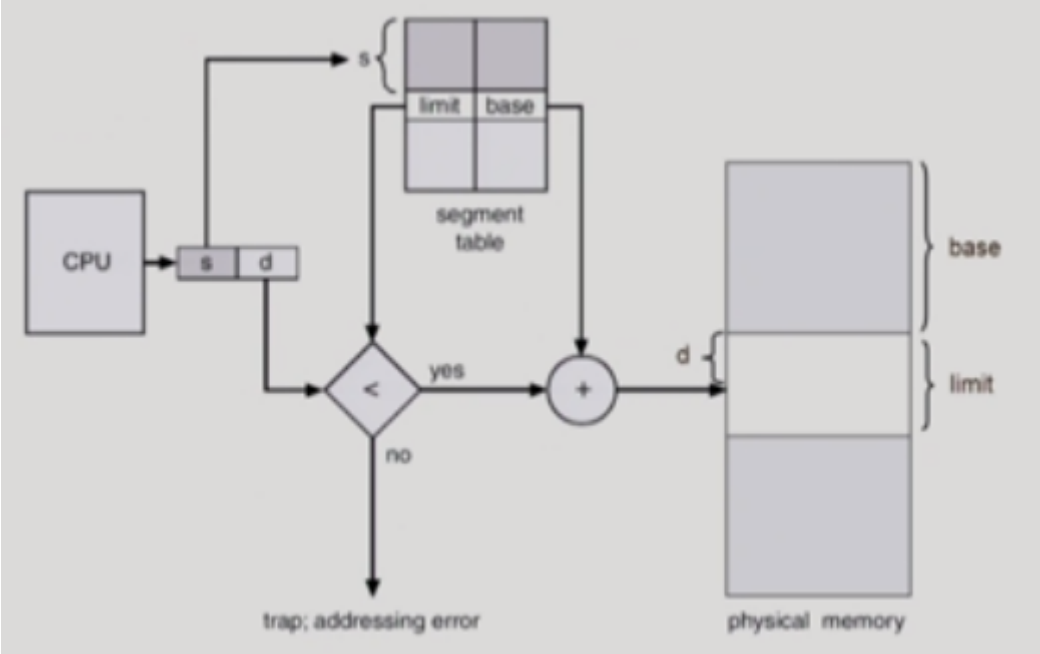
- Shared Page
  - 공유할 수 있는 코드는 같은 프레임으로 매핑 시켜서 올림



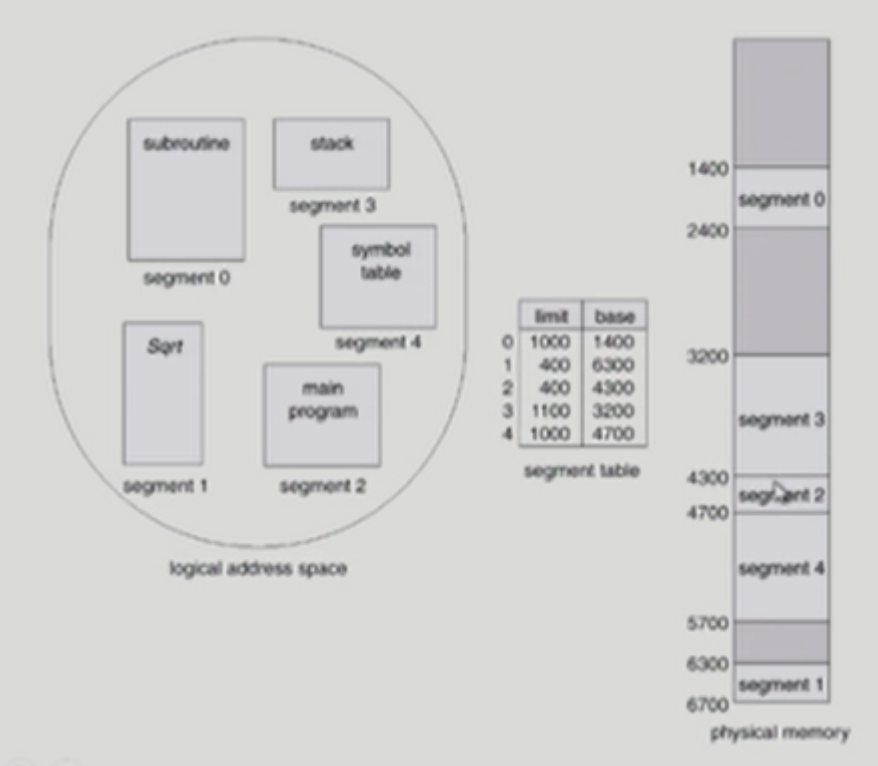
- Shared Code
  - Re-entrant Code(=Pure code)
  - read-only로 하여 프로세스 간에 하나의 code만 메모리에 올림
  - shared code는 모든 프로세스의 logical address space에서 동일한 위치에 있어야 한다.
- Private code and data
  - 각 프로세스들은 독자적으로 메모리에 올림
  - Private data는 logical address space의 아무 곳에 와도 무방

2. Segmentation

- 프로그램은 의미 단위인 여러 개의 segment로 구성
  - 작게는 프로그램을 구성하는 함수 하나하나를 세그먼트로 정의
  - 크게는 프로그램 전체를 하나의 세그먼트로 정의 가능
  - 일반적으로는 code, data, stack 부분이 하나씩의 세그먼트로 정의된다.
  - main(), function, global variables, stack, symbol table, arrays
- Segmentation Architecture
  - Logical address는 다음 두 가지로 구성
    1. segment-number
    2. offset
  - segment table
    - 각각의 테이블 엔트리는 `base`, `limit` 를 가지고 있다. `limit` 는 길이를 `base` 는 물리적 주소의 시작을 나타낸다.
  - segment-table base register(STBR)
    - 물리적 메모리에서의 segment table의 위치
  - segment-table length register(STLR)
    - 프로그램이 사용하는 segment의 수
    - S의 값과 비교함.
  - Segmentataion Hardware



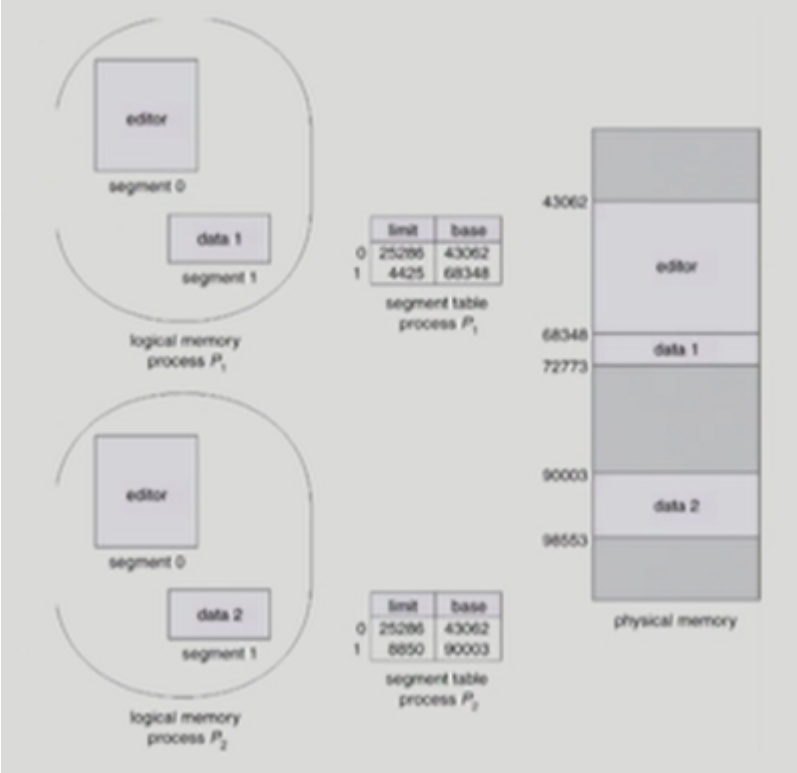
- Example of Segmetation



- Segmentation Architecture(Cont.)

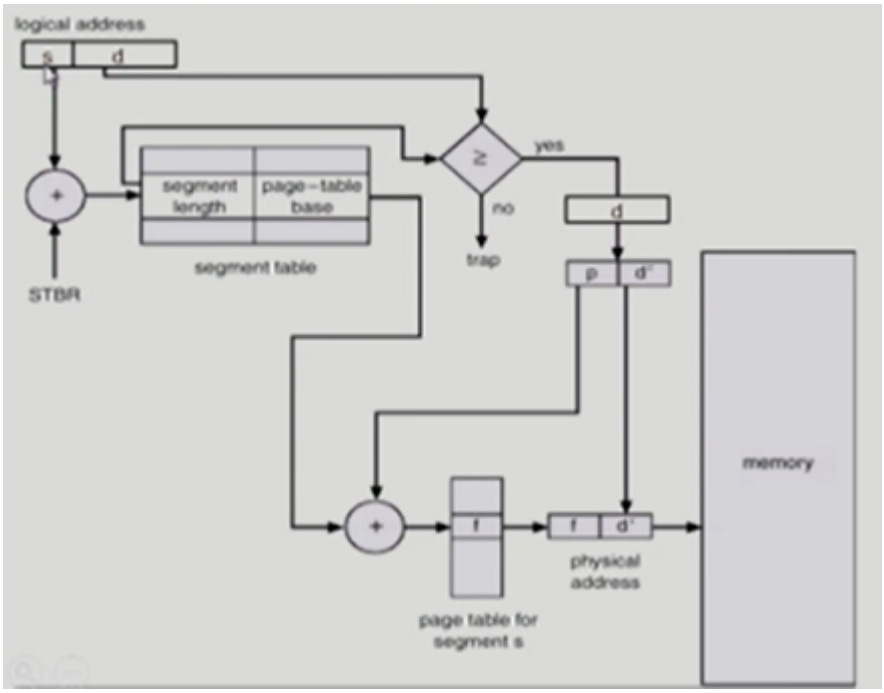
1. Protexction
    a. 각 세그먼트 별로 protection bit가 있다.
    b. Each entry:
        i. Valid bit = 0 → illegal segment
        ii. Read/Write/Execution 권한 bit
2. Sharing
    a. shared segment
    b. same segment number
    c. segment는 의미 단위이기 때문에 공유(sharing)와 보안(protection)에 있어 paging보다 훨씬 효과적이다.
3. Allocation
    a. first fit / best fit
    b. external fragmentation 발생
    c. segment의 길이가 동일하지 않으므로 가변분할 방식에서와 동일한 문제점들이 발생한다.

- Sharing of Segments



code는 현재 shared 상태이다. (0번 세그먼트)

- **Segmentation with Paging**
  - pure segmentation과의 차이점
    - segment-table entry가 segment의 base address를 가지고 있는 것이 아니라 segment를 구성하는 page table의 base address를 가지고 있다.



- Address Translation Architecture
  - 이 챕터에서는 운영체제의 큰 역할을 없다고 보면 된다.