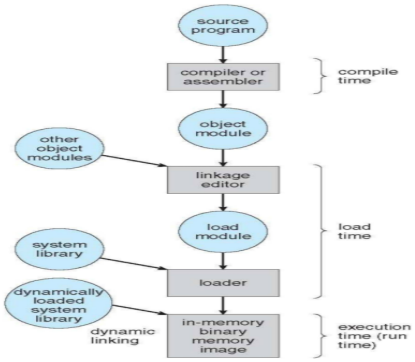
운영체제 Chapter 7

- Memory Management -

1. 배경지식
   - 프로그램은 실행되기 위해 디스크에서 메모리에 위치해 있어야함
     - Main memory는 cpu가 직접 접근할 수 있는 메모리이기 때문에
   - 메모리는 일련의 바이트의 연속으로 바이트 단위별 주소를 가지며, 읽기/쓰기가 가능함
   - 레지스터 접근은 cpu 1 clock 이 소모되거나 덜 걸린다. (버스를 거치지 않기 때문에 빠름)
   - 메모리는 버스를 통해 접근하기 때문에 cpu가 데이터를 사용 가능 할 때까지 stall 이라는 지연이 존재

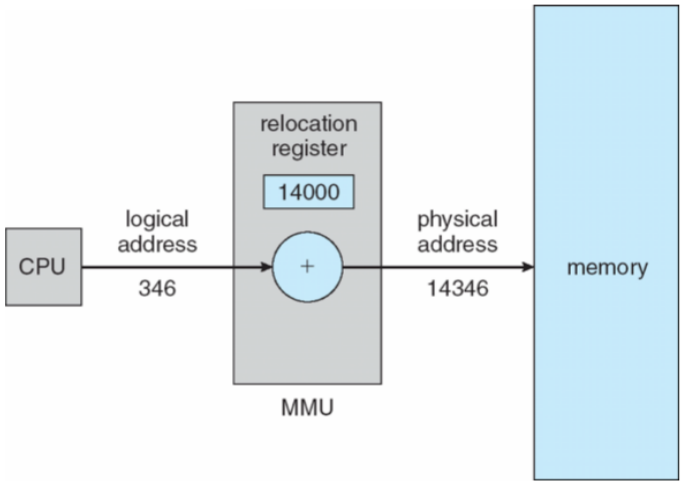
2. Base and Limit Register
   - base / limit register를 사용해서 올바른 메모리 접근을 돕는다.
   - base register : 접근하려는 실행 단위(프로세스나, 명령어)의 논리적 시작 주소를 담고있는 레지스터
   - limit register : 접근하려는 실행 단위의 크기를 담고 있는 레지스터
   - base reg ≤ cpu가 접근하는 주소 < base reg + limit reg
     - 위의 범위를 벗어나면 소프트웨어 인터럽트 발생 (trap)

3. 명령어와 데이터가 메모리에 바인딩 되는 것
   - Compile time
     - 프로세스가 어디에 적재될지 컴파일러가 알고 있으면 주소변환이 컴파일 시에 이루어짐
     - 변경해야 하는 경우 컴파일을 다시 해줘야함
     - 변경될 수 없는 컴파일 된 코드를 absolute code 라고함
   - Load time
     - 프로그램이 실행될 때 즉, 메모리에 로드될 때 최종 물리주소가 결정됨
     - 컴파일러가 재배치 가능한 코드라고 해서 relocatable code 라고함
   - Execution time
     - runtime 바인딩이라고도 불리며, 실행될 때 메모리의 주소가 결정되는 것
     - 프로그램이 실행된 후에도 주소를 변경할 수 있음



4. Logical vs Physical Address Space
   - 별도의 물리적 주소 공간에 바인딩 된 논리 주소 공간의 개념은 메모리 관리에 핵심이다.
   - Logical Address : cpu에 의해 생성되며, 가상 주소라고도 함
   - Physical Address : 논리주소에 대응하는 실제주소가 물리주소이며, 메모리가 취급하는 주소
   - 논리와 물리 주소는 compile time과 load time 바인딩체계에서 동일하나, execution time 바인딩에서 다르다.

5. Memory Management Unit (MMU)
   - 논리 주소(가상 주소)를 물리주소로 매핑(변환)하는 작업을 수행하는 하드웨어 장치
   - base register가 relocation register라고 불리며, 아주 간단한 변환 방식을 사용함

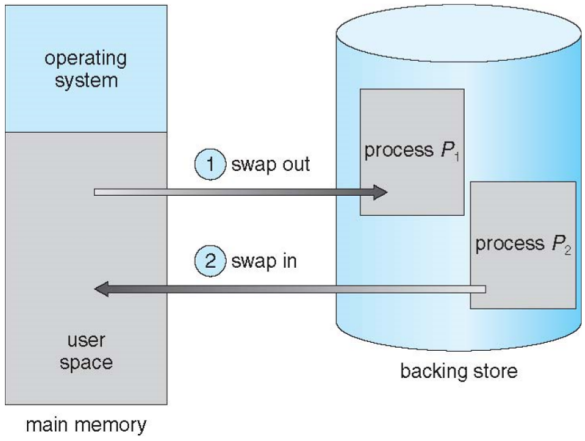


모든 논리 주소에 14000을 더하여 물리주소로 변환하는 방식이며, MMU가 메모리를 접근할 때마다, 또는 실행되는 중간 중간에 이 변환작업을 실행한다.

6. Dynamic Linking
   - Static Linking은 전통적인 방식으로 라이브러리(ex. printf와 같은 라이브러리)를 매 프로세스에 넣는 방식 이지만, Dynamic Linking은 execution time(runtime)까지 linking을 하지 않을 뿐만 아니라 1개의 라이브러리를 동적으로 추가하면 되기 때문에 메모리 낭비를 줄일 수 있는 효과가 있다.
   - 자주 사용되는 라이브러리는 메모리에 상주되어 있고, 프로세스마다 stub을 가지며 해당 stub을 사용하여 라이브러리와 연결시키는 방식
   - 하나의 라이브러리를 공유하기 때문에 서로 다른 프로세스 간에 주소 공간 문제가 발생할 수 있음

7. Swapping

- medium term scheduling에서 사용되는 메모리 관리기법이며, 디스크를 이용해 메모리를 확장시키는 개념으로 메모리에서 프로세스를 디스크로 임시로 쫓아내거나 다시 대려온다.
- backing store
  - 하드디스크, 메모리의 모든 복사본을 수용할 수 있을 만큼 큰 고속 디스크
- Roll out, Roll in
  - 우선순위를 기반으로 swap out, swap in 하는 것을 뜻함
    ex) Roll out : 시스템에 부하가 있을 때 우선순위가 낮을 것을 디스크로 swap out 시킨다.
- 시스템은 swapping 하기 위해 디스크에 있는 프로세스를 관리하는 ready queue를 유지해야함



8. Swapping on Mobile System

- 모바일 시스템은 Swapping 할 여건이 되지 않음
- 모바일 시스템은 flash memory 기반이기 때문에 swapping에 있어 여러 제약이 있음
  - 메모리 접근동작에 있어 비대칭임(쓰기동작이 읽기동작보다 훨씬 느림)
  - 메모리 자체에 수명이 있어 읽기쓰기에 제한이 있음

9. Contiguous Allocation (과거의 OS에서 사용하던 메모리 연속할당 기법)

- main memory는 운영체제뿐만 아니라 여러 사용자 프로세스도 수용해야함
- 하나의 프로세스를 메모리 공간에 연속적으로 저장하는 방법이다.
- 제한적인 자원을 효율적으로 할당하기 위한 초기의 방법 중 하나가 연속할당이었음
- 메모리는 두 개의 파티션으로 나누어진다.
  - 하나는 메모리에 상주하는 운영체제를 위한 것이고, 다른 하나는 사용자 프로세스를 위한 것임
  - 유저 프로세스가 상위 메모리에 보관됨

10. Multiple-partition allocation
- 동시에 실행 가능한 프로세스의 개수는 메모리 파티션의 개수에 의해 제한된다.
- 가변적인 파티션 크기, 프로세스의 필요에 맞게 크기를 조정할 수 있다.
- Hole, 사용가능한 메모리 블록으로서 다양한 크기의 hole이 메모리 전체에 흩어져 있음
- 프로세스를 수용할 수 있을 만큼 충분히 큰 hole에 메모리가 할당됨
- 프로세스가 종료되어, 가용해진 파티션은 인접한 hole과 결합한다.
- OS는 할당된 파티션과, hole의 대한 정보를 계속 알고 있어야함

11. Dynamic Storage-Allocation Problem(hole에서 원하는 크기 n만큼의 요청을 만족시키는 방법)
- First-fit : 충분히 큰 첫 번째 hole에 할당하는 방법
- Best-fit : 요청한 크기보다 크면서 가장 작은 hole에 할당하는 방법
  - 크기별로 정렬되어 있지 않다면, 메모리 전체를 search 해야 함
  - 매우 작은 hole들이 생겨나게 됨 -> 외부 단편화
- Worst-fit : 가장 큰 hole을 할당하는 방법
  - 역시 메모리 전체를 search 해야 함
  - 큰 size의 hole들이 줄어들게 됨

∵ **속도와 스토리지 활용도 측면에서 worst-fit에 비해 first-fit과 best-fit이 더 좋은 방법임**

12. Fragmentation
- External Fragmentation
  - 요청을 충족시킬 만큼의 총 메모리 공간은 존재하지만, free 메모리가 연속적이지 않기 때문에 요청한 만큼의 메모리를 할당할 수 없는 문제
- Internal Fragmentation
  - 할당된 메모리가 요청한 크기만큼의 메모리보다 약간 더 클 수도 있음
  - 할당된 메모리 공간 내에 사용되지 않는 메모리로 인해, 내부 단편화가 발생했다고 함
- Compaction : 작은 hole들을 다 몰아서 큰 hole을 확보하자! 외부 단편화를 줄여보자!
  - 메모리의 모든 내용들을 한군데 몰고, 모든 hole들을 다른 한 군데로 몰아서 큰 블록을 만드는 것.
  - 압축이 일어나기 위해서는 프로세스들의 재배치가 execution time에 동적으로 이루어지는 경우에만 가능하다.
  - 그러나, hole을 움직이거나, 프로세스를 움직이는 Compaction에서의 최적 알고리즘은 존재하지 않다. -> 메모리 계산의 부담이 발생하기 때문에 힘든 작업
  - 그래서 메모리의 낭비 공간인 외부단편화를 최소한으로 만들기 위해 Paging기법이 등장함!

13. Segmentation (가상 메모리 관리 기법)
- 메모리를 바라보는 사용자 관점 그대로 지원한다.
  - 프로세스를 논리적 내용 즉, 세그먼트 단위로 잘라서 메모리에 **연속적으로 배치**
  - 프로세스는 세그먼트의 집합으로서, 세그먼트의 크기는 다양함
    -> 세그먼트의 크기가 서로 다르기 때문에, 내부 단편화는 발생하지 않으나 외부 단편화는 존재
    -> 원하는 세그먼트 크기별로 물리메모리에 할당이 되면서 내부 단편화는 발생하지 않지만, 메모리 hole이 세그먼트보다 작은 경우엔 할당이 불가능하기 때문에 외부 단편화가 발생함

- 여기서 말하는 세그먼트란, 프로세스의 구성 중 code, data, stack이 세 가지는 물론이며 더 들어가 함수, 루틴, 전역변수, 구조체, 배열 등 논리적 내용의 단위를 의미한다.

• Segmentation 기법에서는 논리 주소가 <세그먼트 번호, 오프셋>으로 구성되어, 메모리 내 세그먼트의 위치를 찾을 때 용이함

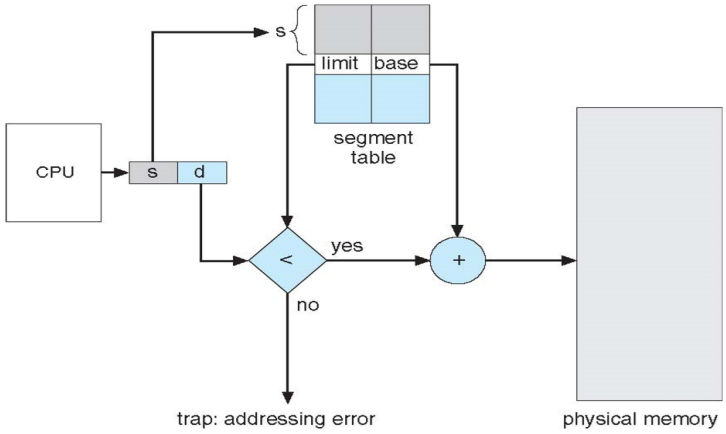
• 논리주소와 물리주소를 매핑하기 위해 Segment table은 세그먼트의 시작주소(base)와 세그먼트의 길이(limit)정보가 담긴 테이블로 운영된다.

• Segment-table base register (STBR)은 세그먼트 테이블의 시작주소를 담고 있는 레지스터

• Segment-table length register (STLR) 프로그램에 사용되는 세그먼트의 수를 담고 있는 레지스터

- 실행하려는 프로세스의 길이를 나타낸다고 해도 무방함
- 세그먼트 번호가 STLR보다 작을 경우에 유효하다.

• MMU가 Segmentation 기법을 사용하는 구조에서 논리주소->물리주소 변환 과정



이것보다 paging에서의 과정이 더 중요함..

14. Paging (가상 메모리 관리 기법)

• 프로세스의 물리 주소공간은 **비연속적으로 할당** 될 수 있으며, 프로세스는 물리적 메모리가 가용될 때 언제든 메모리에 할당할 수 있음

• 메모리를 고정된 크기로 나눈 것을 frame이라고 한다.

- frame의 크기는 작게는 512B부터 1KB, 2KB, … ,16MB까지 2^nByte를 따른다.

• 논리메모리(프로그램, 프로세스)도 같은 frame단위인 page로 잘라서 메모리에 적재하는 방식이다.

- 연속적인 물리메모리가 아니더라도 원하는 크기의 프레임을 사용할 수 있음
  -> 그렇기 때문에 외부 단편화가 발생하지 않으나 내부 단편화는 존재
  -> 프로세스를 고정된 frame크기에 맞게 page로 자르다보면 끄트머리에 frame크기 보다 작은 page가 생겨날 수 있는데 그것이 frame에 할당될 때 내부 단편화 발생한다고 보는 것임

• Paging 기법에서 시스템은 모든 free frame을 추적하고 관리해야함

• N개의 page로 구성된 프로그램(프로세스)를 실행하려면, N개의 free frame을 찾고 프로그램을 로드해야함

• CPU가 생성하는 논리주소와 메모리의 물리주소를 매핑하기 위해 page table이 필요함

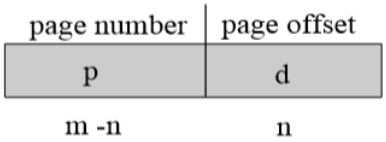
- 각 페이지가 어떤 프레임에 연결되어 있는지 알기 위함

15. Address Translation Scheme (주소변환체계)

• Paging기법에서 CPU에 의해 생성된 논리주소는 두 부분으로 나뉨

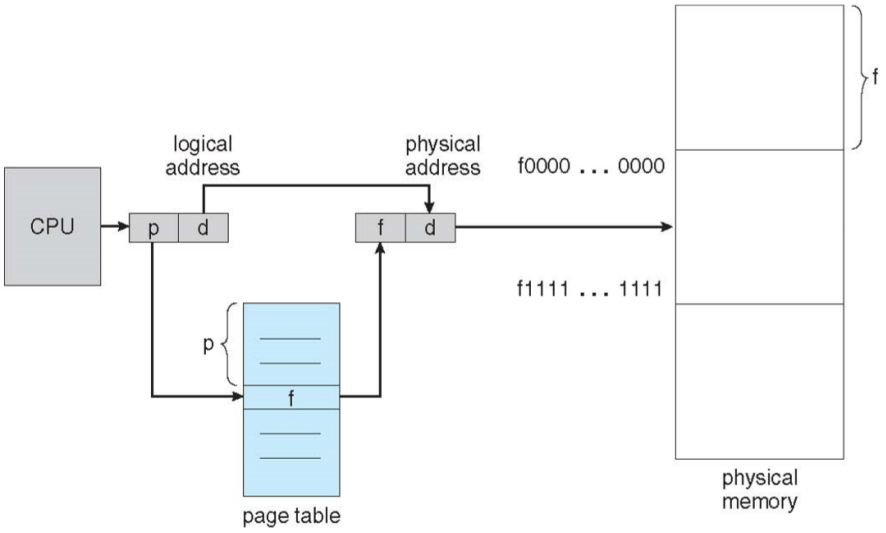
- Page Number(p) : page table에 저장되는 index로 사용되며, 각 page 별로 물리 메모리에서의 시작주소를 담고 있음
- Page Offset(d) : 물리 메모리에서의 해당 p번 페이지의 오프셋을 나타내며, 시작주소와 결합되어 메모리 장치로 보내지는 실제 메모리 주소

- 예를 들어 32bit 주소체계 컴퓨터이고, 페이지 크기를 2^{10}(1KB) 단위로 자른다고 가정해보자.



-> d은 10bit를 사용하고, p의 주소공간은 22bit이다.

- 예를 들어 50번지라는 논리 주소를 물리 주소로 바꾸는 과정을 살펴보자. (페이지 크기는 2^4Byte)



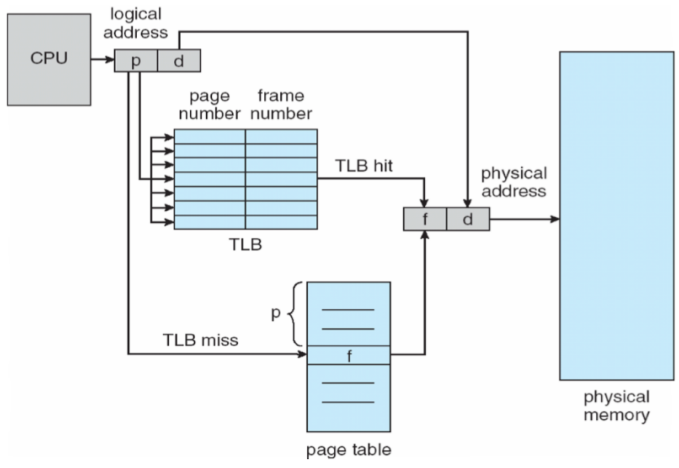
-> 50번지는 2진수로 11 0010이 되는데, 하위 4비트는 offset(d)이며 상위 2비트는 페이지의 번호 즉 페이지 테이블에서의 index이다. 그러면 페이지 테이블에서 상위 2비트 11에 해당하는 3의 값을 가진 페이지 테이블로 이동하여 그 안에 적힌 값을 읽었더니 f라는 값이 나왔다(가정). 이 f는 frame number이며 이진수로 1111이고, 아까 50번지 논리주소 하위 4비트 0010과 결합하여 물리주소 1111 0010이 탄생하게 되는 것이다. 10진수로 바꾸면 242번지가 되는 셈이다.
결론적으로, 논리주소 50번지는 물리주소 1111 0010이며 f에서의 240번지에다가 offset 0010을 적용하여 242번지 위치라는 것을 알 수 있다.

-> 더 빨리 구하는 방법, 페이지 크기가 16B이므로 50번지를 16으로 나눈 몫은 3이 나오고 이 3은 페이지 테이블의 index이며 3에 해당하는 frame number인 f를 구할 수 있다.(가정)
그리고 50번지를 16으로 나눈 나머지는 2이며, 이는 offset값에 해당한다.
f를 2진수로 1111과, offset을 2진수로 0010(4bit이기 때문에) 결합하여 1111 0010이 탄생!

16. 내부 단편화를 계산하는 방법
- Page의 크기 = 2,048 bytes
- Process의 크기 = 72,766 bytes
- Page의 수 = 72,766 / 2,048 = 35 pages + 1,086 bytes
- 남은 1,086 bytes도 2,048 bytes에 할당해야함
- 2,048 bytes - 1,086 bytes = 962 bytes
- 즉, 962 bytes 만큼의 내부 단편화 발생
- 가장 최악의 경우는 2,048 bytes에 1 bytes를 할당해야 하는 상황, 2,047 bytes의 내부 단편화
- 평균적으로 프레임의 1/2 사이즈 정도 내부단편화가 발생함
- 내부 단편화를 줄이고자 frame size를 줄이면, page의 수가 많아지면서 page table이 커지게 되고 메모리 낭비로 이어진다. -> Trade off(절충)이 필요함

17. Implementation of Page Table
- Page table은 프로세스마다 존재하는 자료구조임
- Page table은 메인 메모리에 보관됨
- Page-table base register (PTBR)는 페이지 테이블의 시작주소를 담고 있는 레지스터
- Page-table length register (PTLR)은 페이지 테이블의 크기를 나타내는 레지스터
  - 메모리에 적재하려는 프로세스의 페이지의 수 라고 해도 무방함
- Paging 방식에선 메인 메모리에 있는 Page Table에 접근하고, 데이터를 얻기 위해 물리 메모리에도 접근하여 메모리 접근을 최소 두 번 해야 하기 때문에 성능이 떨어질 수 있다.
  - 더군다나 Page table은 매우 자주 사용되는데, page table에서도 접근 빈도가 높은 일부분을 associative memory 또는 TLBs 캐시에 일부만 적재하여 접근 효율을 높일 수 있다.
- TLBs는 associative memory의 한 종류이기 때문에 검색시간이 늘 일정하다.

18. TLB를 사용한 Paging 동작과정



1) p를 이용해 TLB를 가장 먼저 탐색한다.
2) TLB hit면 굉장히 빠른 속도로 주소변환이 일어나고, 보다 빨리 물리 메모리에 접근할 수 있다.
3) TLB miss면 원본 데이터, Page table을 탐색하여 주소변환 과정을 거친 후 물리메모리에 접근한다.
   -> TLB miss면 TLB를 도입하지 않은 것보다 주소변환이 오래 걸리며, 물리 메모리 접근에도 더 느려진다.

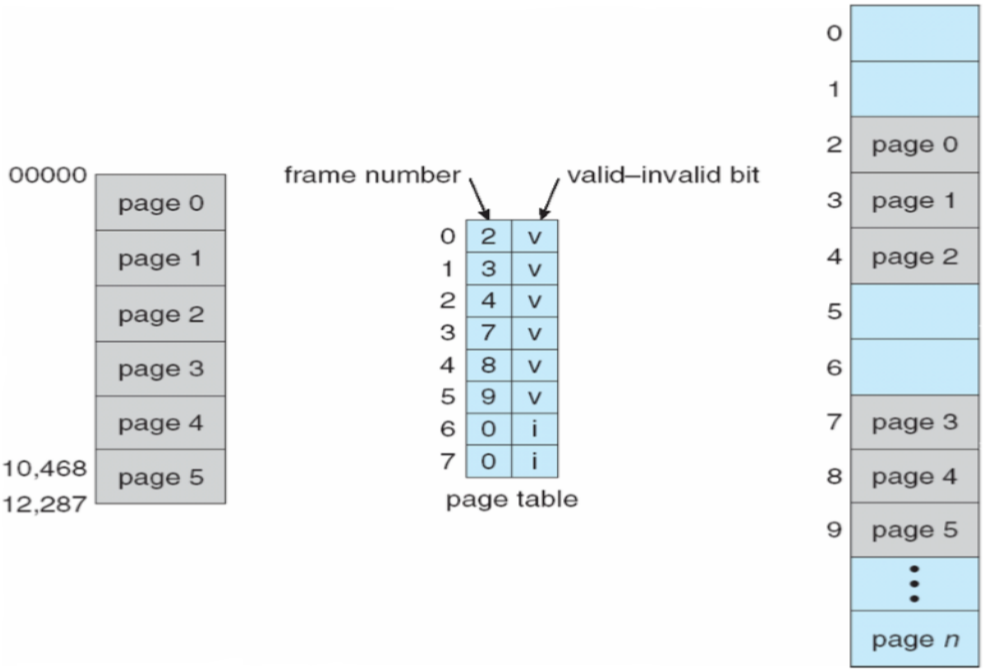
19. Effective Access Time (유효 메모리 접근시간)

- Associative 검색, TLB 접근시간 = e 정도 걸린다고 가정
  - 메모리 접근 시간의 10% 보다도 덜 걸릴 수 있음
- 메모리 접근시간을 1이라고 가정
- TLB에 page number가 발견 될 확률, hit ratio = a (0 < a < 1)
- 유휴 메모리 접근시간(EAT) = TLB hit 발생 시 전체 접근시간 + TLB miss 발생 시 전체 접근시간

  = (1 + e) * a + (2 + e)(1 - a)

  = **2 + e - a**
- a = 80%, e = 20ns, 메모리 접근시간을 100ns 라고 가정하면
  - EAT = 200ns + 20ns - 80ns = 140ns
- 더 현실적인 가정으로 a = 99%, e = 20ns, 메모리 접근시간 100ns
  - EAT = 200ns + 20ns - 99ns = 121ns

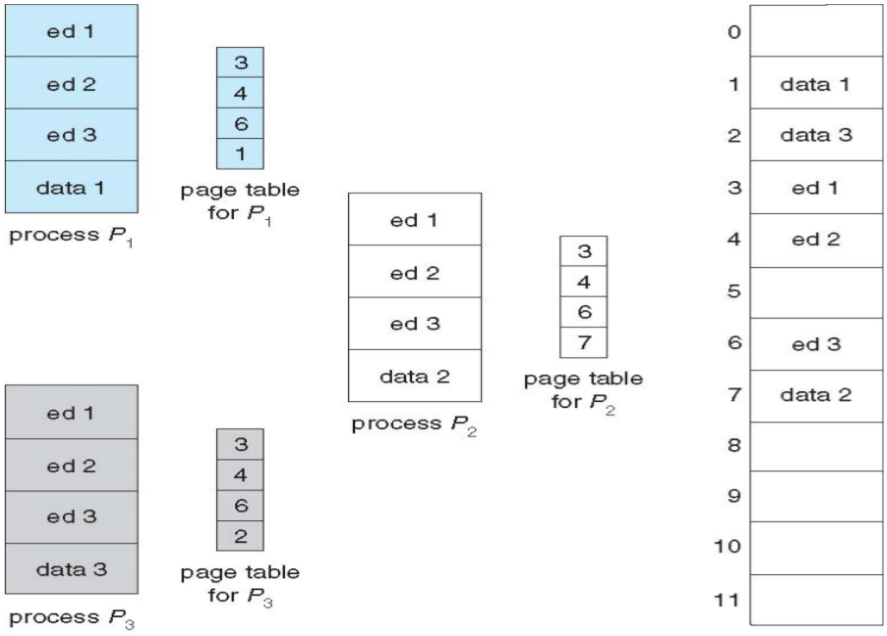
20. Memory Protection



- page table에 있는 모든 entry에 valid-invalid bit를 생성한다.
  - valid : logical memory에 page가 존재한다는 의미로, 주소변환에 유효한 페이지임을 뜻함
  - invalid : logical memory에 page가 존재하지 않는다는 의미로, 잘못된 접근임을 뜻함

    -> 접근 시 소프트웨어 인터럽트(trap)이 발생

21. Shared Pages (그냥 간단하게 이해하고 넘어감)



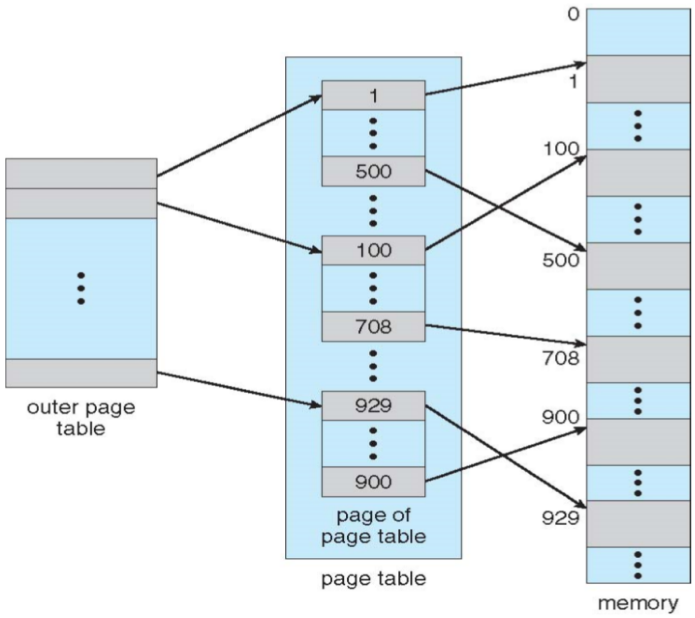
- 3개의 프로세스 간 공유하고 있는 ed1, ed2, ed3는 read-only 코드임
  - 각기 다른 logical memory에 존재하지만 같은 frame에 mapping 될 수도 있다!
- data1, data2, data3 은 private code and data라고 하며 각자의 logical memory에 있음

22. Structure of the Page Table

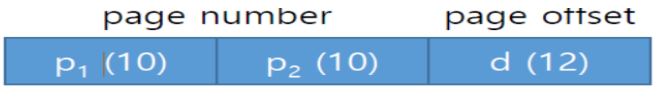
- 32비트 주소체계 컴퓨터가 있다고 가정해보자
  - page size는 4KB(2^{12}) 일 때, page table의 수는 $2^{32} \div 2^{12} = 2^{20,}$ 약 100만개 이다.
  - 또한, 각 entry를 4 bytes라고 할 때, 페이지 테이블 메모리 공간만 총 4MB가 되는셈이다.
  - 이를 연속적으로 주 메모리에 할당하기는 쉬운 작업이라고 할 수 없다. (여러 측면에서..)
  - 페이지 테이블을 할당하는 다양한 방법들을 살펴보도록 하자

23. Hierarchical Paging

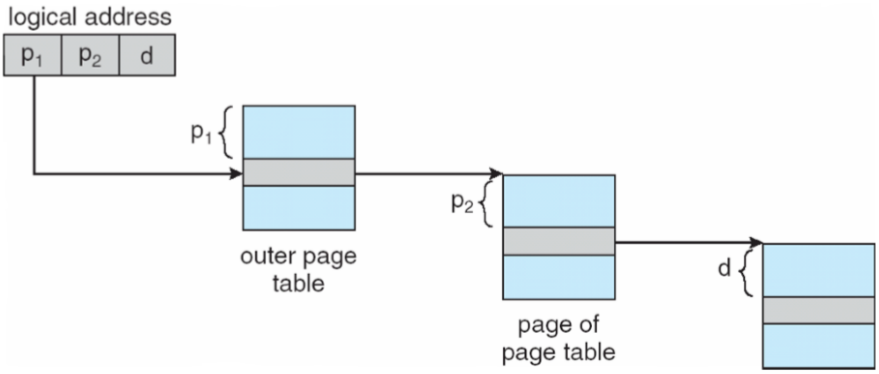
- 계층 구조를 갖는 Paging 기법
- 하나의 logical address를 여러 개의 page table을 통해 여러 level로 나누어 주어 page table의 사이즈를 줄이기 위한 방법이며, 간단한 방법으로는 아래 그림과 같이 Two-level page table이 있다.



- 32비트 주소체계 기계에서, page 한 개의 size는 4KB라고 했을 때,
  - page number(p)를 구성하는 비트는 20bit, offset(d)은 12bit이다.
  - page number(p)를 구성하는 상위비트 20bit중 10bit를 빼서 p2로 사용하고, 나머지를 p1이 사용



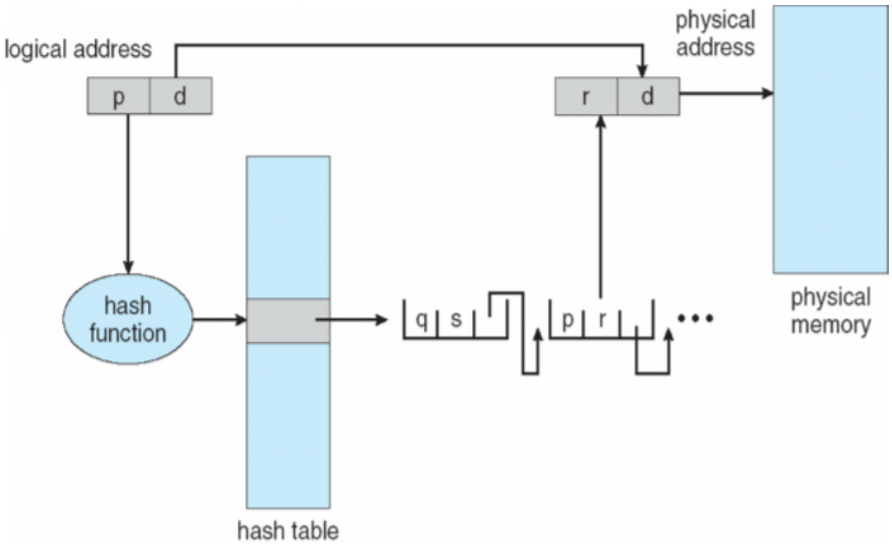
  - outer page table의 엔트리 개수 2^{10}개의 inner page table중 **사용 중인 table만** 메모리에 적재됨



  - 대충 그림보고 이해하자면, p1은 outer page table을 탐색하는데 사용되며, inner page table의 시작주소를 담고 있다.

24. Hashed Page Tables

• Hashing 기법을 사용해 page table을 관리하는 방법



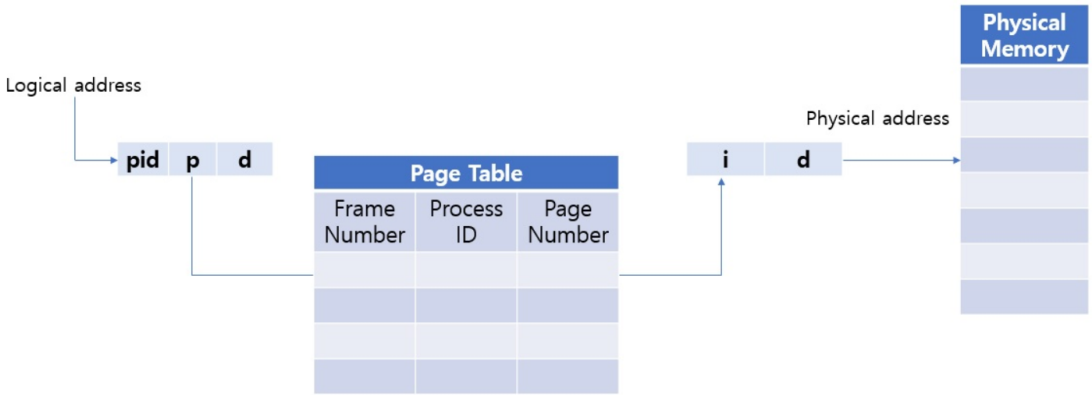
- 논리 주소에서 page number에 해시함수를 적용하여 그 값으로 hash 형태의 page table을 접근함
- 접근 후, 해싱 된 page number와 매칭 되어 있는 frame number를 알려준다.

25. Inverted Page Tables (역전 구조)

• 각각의 프로세스가 page table을 하나씩 가지면, 전체적으로 사용되는 page table의 사이즈가 너무 커지기 때문에, 시스템 전체적으로 페이지 테이블이 무조건 하나만 존재하도록 하는 구조

• 이전의 table들은 모두 Logical address를 기준으로 하는 page table이었음

- 나누어진 페이지의 순서대로 Page table에 순서대로 대응되는 구조였다.
- 그러나, 역전 구조는 그 반대로 Physical address의 입장에서 만들어진 테이블 구조임
- 첫 번째 frame은 table의 첫 번째에 저장되며 table은 각 frame이 어떤 프로세스와 대응되는지에 대한 정보를 담고 있는 구조! -> **메모리에서 훨씬 작은 공간을 점유하는 장점!**



- 논리 주소는 <process-id, page-number, offset> 세 가지 항목으로 구성됨
- process ID를 찾기 위해서 전체의 table을 검색해야 한다는 단점이 있다. (탐색시간↑)