8. Memory Management

▼ 목차

1. Memory

1-1. 논리적 주소

프로세스마다 독립적으로 가지는 주소 공간
각 프로세스마다 0번지부터 시작
CPU가 바라보는 주소

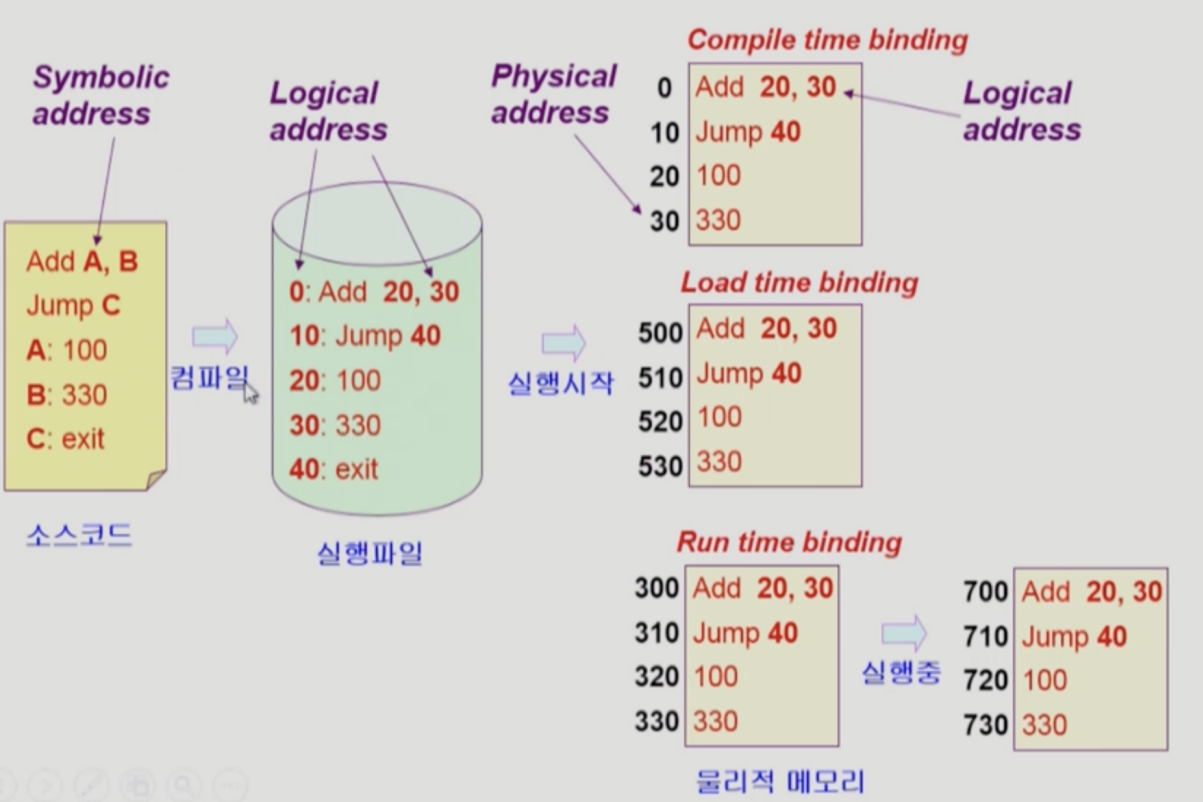
1-2. 물리적 주소

메모리에 실제 올라가는 위치
프로세스가 실행되기 위해 실제로 메모리(RAM)에 올라가는 위치

1-3. 주소 바인딩

어떤 프로그램이 메모리의 어느 위치에, 즉 어떤 물리적 주소가 결정되는 과정
Symbolic address → Logical address → Physical address

주소 바인딩이 일어나는 시점 3가지
논리적 주소에서 물리적 주소로 넘어가며 결정되는 시점



1. Compile Time binding

프로세스의 물리적 주소가 컴파일 때 정해짐
프로세스가 메모리의 어느 위치에 들어갈지 미리 알고 있다면 컴파일러가 절대 주소 (Absolute address),
즉 고정된 주소를 생성한다. 따라서 만약 위치가 변경된다면 재컴파일을 해주어야 한다.
프로그램이 실행될 때 이미 주소가 결정되고 변하지 않는다
컴파일 타임 주소 할당은 프로세스 내부에서 사용하는 논리적 주소와 물리적 주소가 동일하다.

컴파일 타임 주소 할당의 문제점은, 주소가 고정되어 있기 때문에 메모리 상에 빈 공간이 많이 발생할 수 있어 비효율적이고, 로드하려는 위치에 이미 다른 프로세스가 존재할 수 있다.

2. Load Time binding

프로세스가 메모리의 어느 위치에 들어갈지 미리 알 수 없다면 컴파일러는 Relocatable code를 생성해야 한다. Relocatable code는 메모리의 어느 위치에서나 수행될 수 있는 기계 언어 코드이다.
그리고 **Loader가 프로세스를 메모리에 load 하는 시점에 물리적 주소를 결정한다.**
실행시 비어있는 위치에 어디든지 올라갈 수 있다.
따라서 로드 타임 주소 할당은 논리적 주소와 물리적 주소가 다르다.
프로그램이 실행될 때 이미 주소가 결정되고 변하지 않는다

하지만, 프로세스 내에 메모리를 참조하는 명령어들이 많아서 이들의 주소를 다 바꿔줘야 하기 때문에, 로딩할 때의 시간이 매우 커질 수 있다는 단점이 있다.

따라서 컴파일 타임과 로드 타임 주소 할당은 실제로 잘 쓰이지 않는다.

3. Execution Time binding (Run time)

프로세스가 수행이 시작된 이후에 프로세스가 실행될 때 메모리 주소를 바꾸는 방법
즉, **Runtime때 물리적 주소가 결정되며 실행 도중에 주소가 바뀔 수 있다.** CPU가 주소를 참조할 때마다 address mapping table을 이용하여 binding을 점검한다.

런타임 주소 할당은 **MMU(Memory Management Unit)**라는 하드웨어 장치를 사용하여 논리적 주소를 물리적 주소로 바꿔준다. 프로세스가 CPU에서 수행되면서 생성해내는 모든 주소값에 대해서 base register의 값을 더해주어 물리적 주소를 생성하는 방식이다. base register는 하나이므로 프로세스끼리 공유한다.

아래의 그림은 MMU가 어떤 방식으로 프로세스의 논리적 주소를 물리적 주소로 변환되는지를 보여준다.

---

some Terminologies

Dynamic Loading

프로세스 전체를 메모리에 미리 다 올리는 것이 아니라 해당 루틴이 불려질 때 메모리에 load하는 것
좋은 프로그램은 방어적으로 설계되기 때문에 오류 처리 루틴이 굉장히 많다. 그런데 실제로 프로그램에서 자주 사용되는 코드는 한정적이다. 이럴 경우, Dynamic Loading을 사용하면 효율적이다.

memory utilization(메모리 이용도)의 향상

운영체제의 특별한 지원 없이 프로그램 자체에서 구현 가능(OS는 라이브러리를 통해 지원 가능)

- paging 기법과 dynamic loading은 원래는 다른 것이다.
- 프로그래머가 명시적으로 dynamic loading을 구현하는 것이 dynamic loading이지만 운영체제가 알아서 올려놓고, 쫓아내고 하는 것도 dynamic loading이라는 말을 쓰기도 함

Loading: 메모리로 올리는 것을 의미

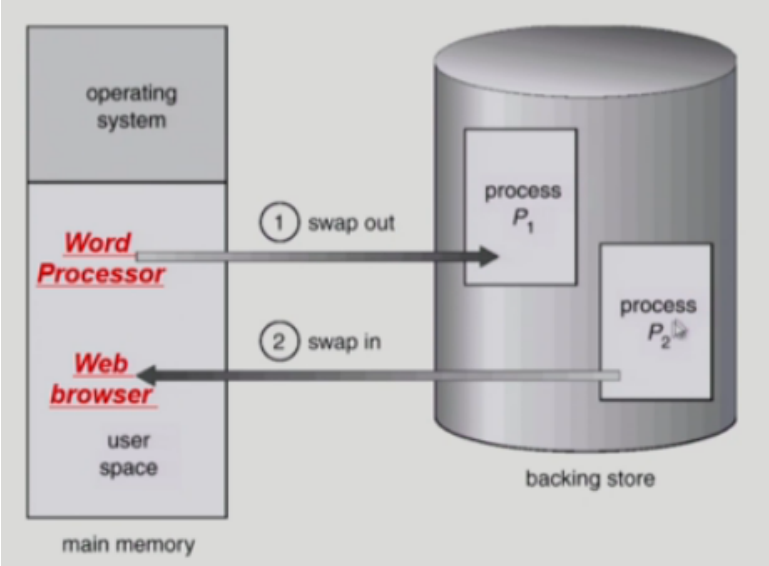
Overlays

메모리에 프로세스의 부분 중 실제 필요한 정보만을 올려놓는 것
운영체제의 지원 없이 사용자에 의해 구현

Dynamic Loading과 다른점
운영체제의 지원이 없고 사용자가 코딩을 통해 구현
작은 공간의 메모리를 사용하던 초창기 시스템에서 수작업으로 프로그래머가 구현

- Manual Overlay
- 운영체제의 라이브러리를 사용할 수 있는 dynamic loading보다 구현이 더 어려움

Swapping



swapping

프로세스를 일시적으로 메모리에서 backing store(하드디스크)로 쫓아내는 것

backing store(=swap area)

디스크: 많은 사용자의 프로세스 이미지를 담을 만큼 충분히 빠르고 큰 저장공간

swap in / swap out

일반적으로 중기 스케줄러(swapper)에 의해 swap out 시킬 프로세스 선정

priority-based CPU scheduling algorithm

- priority가 낮은 프로세스를 swapped out 시킴
- priority가 높은 프로세스를 메모리에 올려 놓음

swapping이 효율적이려면 run time binding이 지원되는 것이 좋음

- compile time 혹은 load time binding에서는 쫓겨난 후에 다시 돌아올 때 원래 메모리 위치로 swap in 해야해서 비효율적임

swap time은 대부분 transfer time(swap되는 양에 비례하는 시간)임

- 용량이 방대하기 때문

지금 우리가 다루는 swapping은 하나의 프로그램이 통째로 쫓겨나는 것을 말한다(원칙적으로는 이게 맞음). 최근에는 page를 활용해서 프로그램 주소공간이 잘게 짤려서 일부 page가 메모리에서 쫓겨나는 형태를 띄는데, 이를 swap out이라고 표현하기도 한다.

Dynamic Linking

linking?

- **여러 군데에 존재하던 컴파일된 파일을 묶어서 하나의 실행파일을 만드는 과정**

dynamic linking: linking을 실행시간(execution time)까지 미루는 기법

Static linking

- 라이브러리가 프로그램의 실행 파일 코드에 포함됨
- 실행 파일 크기가 커짐
- 동일한 라이브러리를 각각의 프로세스가 메모리에 올리므로 메모리 낭비

Dynamic linking

- 라이브러리가 실행시 연결(link)됨
- 라이브러리 호출 부분에 라이브러리 루틴의 위치를 찾기 위한 stub이라는 작은 코드를 둠
- 라이브러리가 이미 메모리에 있으면 그 루틴의 주소로 가고 없으면 디스크에서 읽어옴
- 운영체제의 도움이 필요

물리적인 메모리를 어떻게 관리할 것인가?

---

3. Allocation of Physical Memory

물리적 메모리를 어떻게 관리할 것인가

(1) **OS 상주 영역_(kernel 영역)**

- interrupt vector와 함께 낮은 주소 영역 사용
- 운영체제 커널이 상주

(2) **사용자 프로세스 영역_(user 영역)**

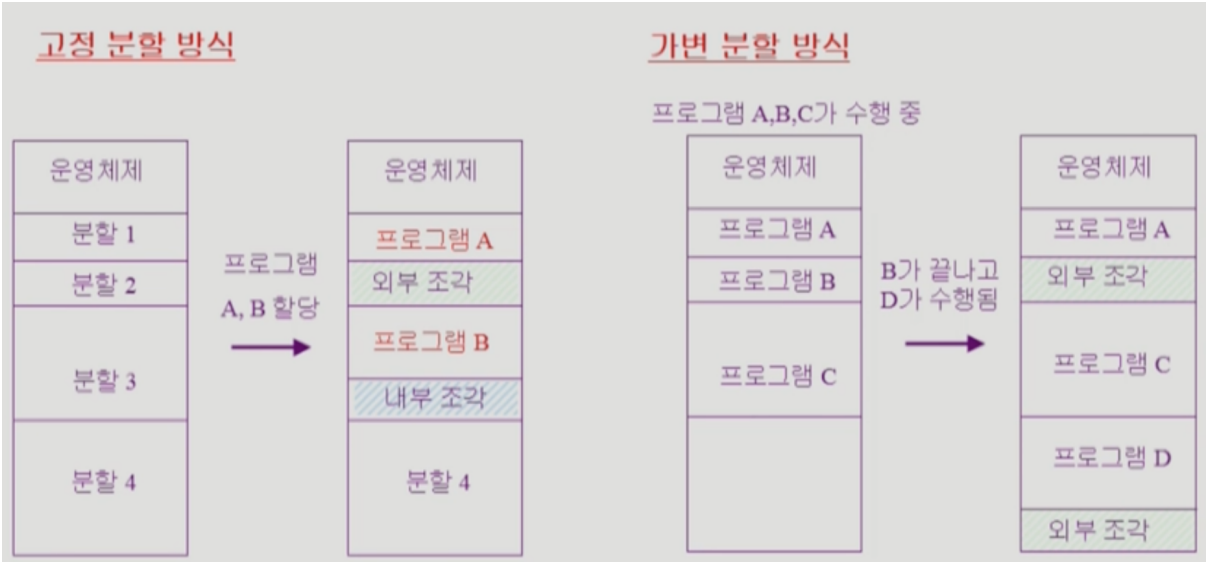
- 높은 주소 영역 사용
- 사용자 프로그램들이 올라가 있음

4. (2) 사용자 프로세스 영역의 할당 방법

4-1. Contiguous allocation(연속 할당)

각각의 프로세스가 메모리의 연속적인 공간에 적재되도록 하는 것

- Fixed partitioin allocation
- Variable partition allocation



고정분할(Fixed partition) 방식

사용자 프로그램이 들어갈 메모리 영역을 미리 나눠놓는 것 (융통성 없음)

internal fragmentation(내부 조각), external fragmentation(외부 조각) 발생 가능
메모리의 낭비가 발생할 수 있다

가변분할(Variable partition) 방식

사용자 프로그램이 들어갈 메모리 영역을 미리 나눠놓지 않고 메모리에 차곡차곡 올리는 것

프로그램의 크기를 고려해서 할당

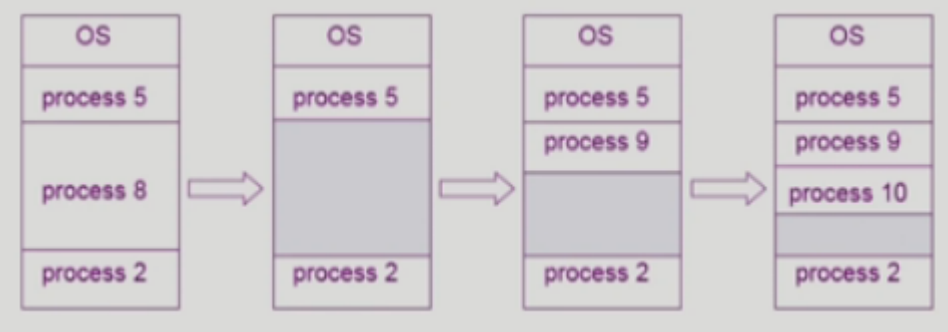
분할의 크기, 개수가 동적으로 변함

프로그램 실행 종료 과정에서 external fragment(외부 조각) 발생

---

hole : 가용 메모리 공간

다양한 크기의 hole들이 메모리 여러 곳에 흩어져 있음
프로세스가 도착하면 수용가능한 hole을 할당
운영체제는 할당공간, 가용공간(hole)을 check하고 있음



Dynamic Storage-Allocation Problem

가변 분할 방식에서 size n인 요청을 만족하는 가장 적절한 hole을 찾는 문제 (n: 프로그램 크기)

(1) **Frist-fit**

Size가 n 이상인 것 중 처음으로 발견되는 hole에 할당

(2) **Best-fit**

Size가 n 이상인 가장 작은 hole을 찾아서 할당
hole들의 리스트가 크기순 정렬되어있지 않으면 모든 hole의 리스트를 탐색해야 함
많은 수의 아주 작은 hole들이 생성됨

(3) **Worst-fit**

가장 큰 hole에 할당
모든 리스트 탐색해야 함
상대적으로 아주 큰 hole들이 생성됨
실험적으로 First-fit, Best-fit이 속도와 공간 이용률(memory utilization) 측면에서 효과적

Compaction

외부 조각으로 인해 생기는 hole들을 한군데로 밀어서 큰 hole을 만드는 것

external fragmentation 문제를 해결하는 방법 중 하나

사용 중인 메모리 영역을 한군데로 몰고 hole들을 다른 한 곳으로 몰아서 큰 block을 만드는 것

매우 비용이 많이 든다.

run time binding이 지원이 되어야 쓸 수 있음

4-2. Noncontiguous allocation(불연속 할당)

하나의 프로세스가 메모리의 여러 영역에 분산되어 올라갈 수 있음

- Paging
- Segmentation
- Paged Segmentation
- 현대 시스템에서는 이 방식 사용

Multilevel paging and performance

- 주소공간이 더 커지면 페이지 테이블을 2단계가 아닌 그 이상의 단계로도 쓸 수 있다.
- 하지만 페이지 테이블이 여러개 만들어지면 주소변환을 위해 메모리에 여러번 접근해야 한다. 시간 오버헤드가 걸릴 수 있다.
- 그러나 대부분의 주소변환이 TLB에 있는 캐싱 정보를 통해 직접 이루어지므로 시간이 지나치게 오래걸리지 않는다.
- 위의 식은 TLB를 사용하면 그 시간이 얼마 걸리지 않음을 나타낸다. 2퍼센트는 TLB가 안되고 98퍼센트는 TLB가 된다. 결과적으로 주소변환을 위해 28ns만 소요된다.

• 프로그램마다 주어지는 로지컬 메모리가 왼쪽, 가운데는 페이지 테이블, 오른쪽은 피지컬 메모리다. 페이지 테이블에는 Valid(v)/invalid(i) 정보가 적혀 있다. v로 적힌 엔트리는 메모리에 올라와 현재 사용되는 영역, i로 적힌 사용되지만 메모리에 올라와있지 않고 swap area(백킹 스토어)에 있는 경우, 엔트리는 존재하지만 사용되지 않는 영역을 의미한다. 프로그램이 가진 주소공간 만큼 페이지 테이블이 존재하기 때문에 사용하지 않는 영역도 엔트리가 만들어지는데, 이를 표시하기 위해서다.

Paging

Process의 virtual memory를 동일한 크기의 page단위로 나눈다.

Virtual memory의 내용은 page 단위로 불연속적으로 저장된다.

일부는 backing storage에, 일부는 physical memory에 저장될 수 있다.

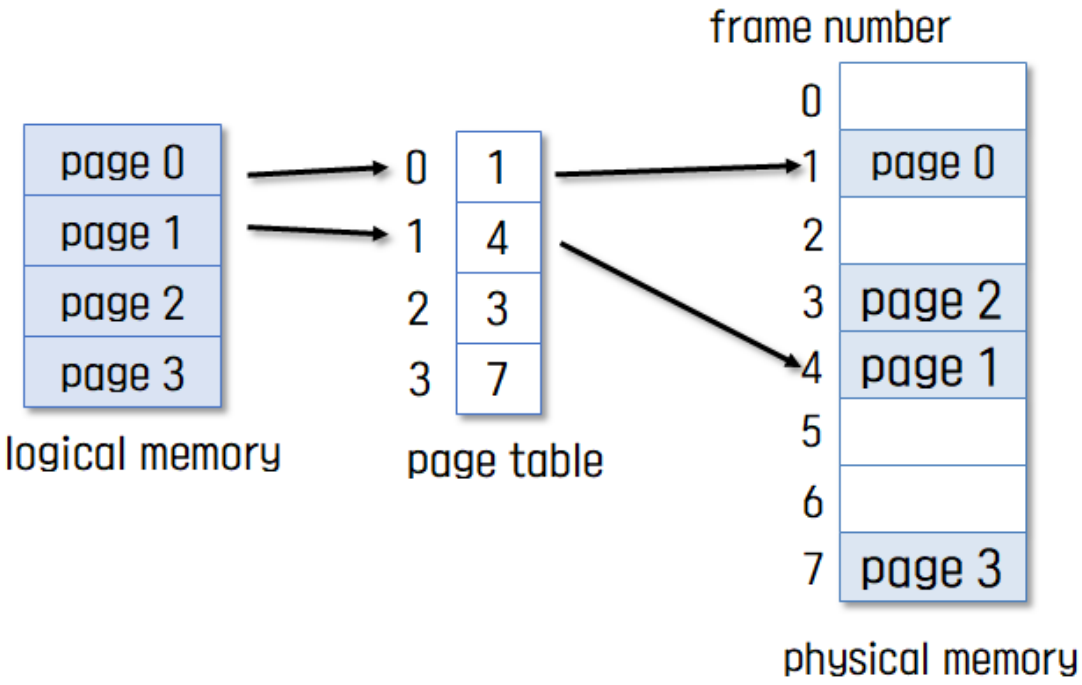
Physical memory를 동일한 크기의 frame으로 나누고, logical memory를 (frame과)동일한 크기의 page로 나눈다.

주소 변환은 page table을 사용하여 logical address를 physical address으로 변환한다.

External fragmentation은 발생하지 않는다.

Internal fragmentation은 발생할 수 있다. **(page frame의 크기 > 마지막 page의 크기)**

Paging 예

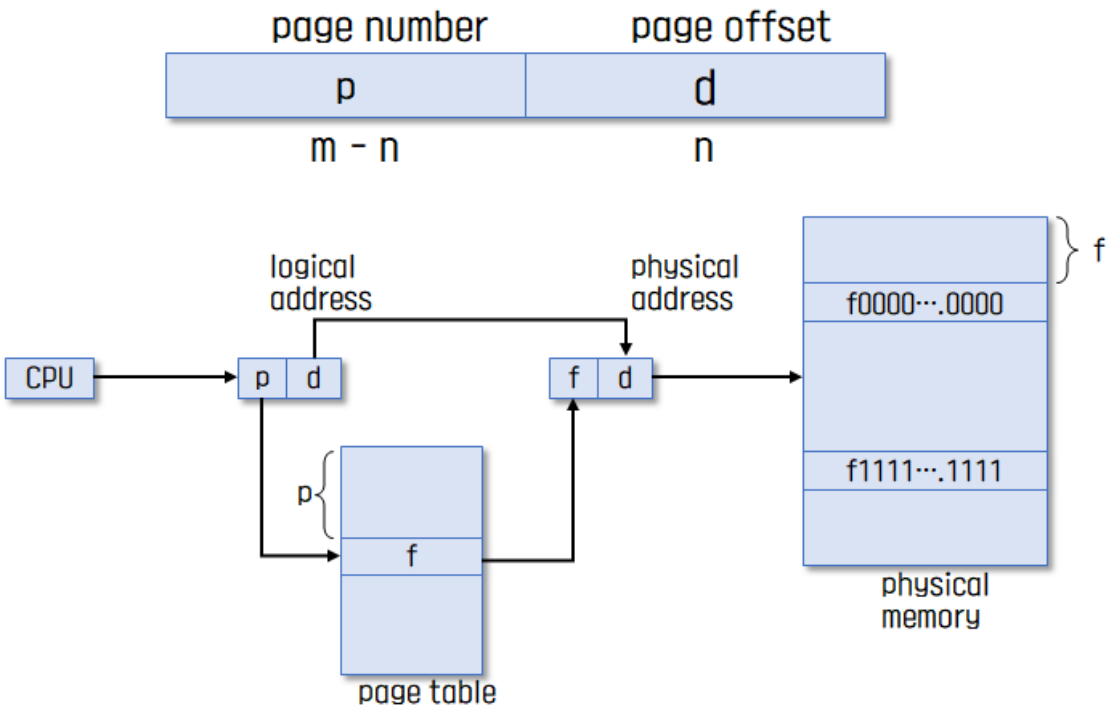


주소 변환을 위해 page table을 활용한다.

logical address가 테이블의 인덱스(entry)를 가리키고,

테이블의 value 값으로 physical address의 frame number 를 갖는다.

Address Translation Architecture



CPU가 다루는 logical address는

- **page number(p)**와
- **page offset(d)**로 나뉜다.

page 번호를 통해 page table에서 frame number를 가져오고,

offset을 통해 해당하는 frame에서 얼만큼 떨어져있는지, 즉 내부에서 위치가 어딘지를 알 수 있다.

보통 p = 20, d = 12 이다.

밑에 다시 한 번 언급하겠지만, 정리하는 겸 써본당.

32bit 주소 체계 기준 보통 페이지의 크기를 **4KB**로 나눈다. 즉, 한 페이지 내부에서 위치를 찾으려면 2^12만큼의 정보를 갖고 있어야 하기 때문에 **12bit**가 필요한 것이다.

마찬가지로 32bit 주소 체계에서의 페이지 테이블의 엔트리 하나의 크기는 32bit, 즉 4B이다. 총 4GB만큼의 정보를 갖고 있어야 하기 때문이다. 그리고 페이지 테이블 하나의 엔트리의 개수는 4GB(메모리의 크기)/4KB(페이지의 크기) = 1M개가 된다. 1M = 2^20이므로, p는 20이 필요하다.

그럼 page table은 어디에 위치하고 있을까?

기초적인 MMU는 레지스터 두 개를 이용해서 주소를 변환했다.

프로그램 하나당 매우 많은 수의 페이지로 나뉠 수 있고, 그렇게 되면 table의 크기도 매우 커질 것이다. 또한, 프로그램마다 각각 별개의 page table을 갖고 있어야 한다. CPU 내부에

레지스터를 두어 page table 정보를 저장하는 것은 현실적으로 불가능하다.

따라서 page table은 **Main Memory**에 상주한다.

기존의 두 개의 레지스터(base, limit)는 다음과 같은 역할을 하게 된다.

- Page-Table Base Register(PTBR) - page table의 시작 위치를 가리킨다.
- Page-Table length Register(PTLR) - 테이블의 크기를 보관한다.

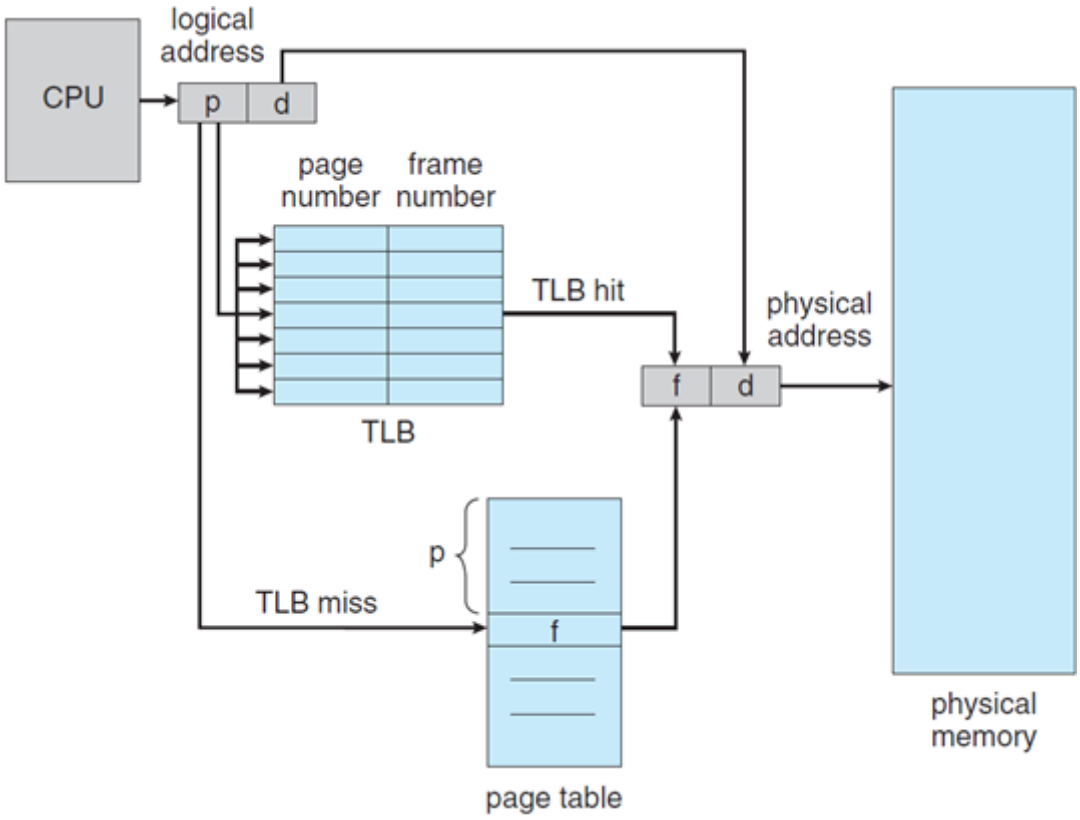
메모리에 접근하려면 page table에 접근해야 하고, 이 page table 또한 메모리에 있기 때문에 **결국 모든 메모리 접근 연산에는 2번의 memory access가 필요**하다.

속도 향상을 위해 별도의 하드웨어를 사용한다.

Translation Look-aside Buffer(TLB)(= associative register) 라는 일종의 캐시를 사용한다.

메인 메모리와 CPU 사이에 존재한다.

Paging Hardware with TLB



TLB는 주소 변환을 위한 캐시 메모리인 셈이다.

page table에서 빈번히 참조되는 일부 엔트리를 캐싱하고 있다.

그래서 page table에 접근하기 전에 TLB에 저장이 되어 있는지 먼저 확인하게 된다.

단, page table의 모든 내용을 담고 있지 않기 때문에 TLB에는 page number와 frame number를 가져야 한다.

또한 TLB에서 해당하는 page table의 엔트리를 찾으려면 O(n) 타임이 걸리게 된다. 따라서 parallel search가 가능한 Associative register를 활용해 탐색한다.

page table과 마찬가지로 TLB도 프로세스마다 다른 정보가 들어 있어야 하기 때문에, context switch 때 flush가 된다.

Two-Level Page Table

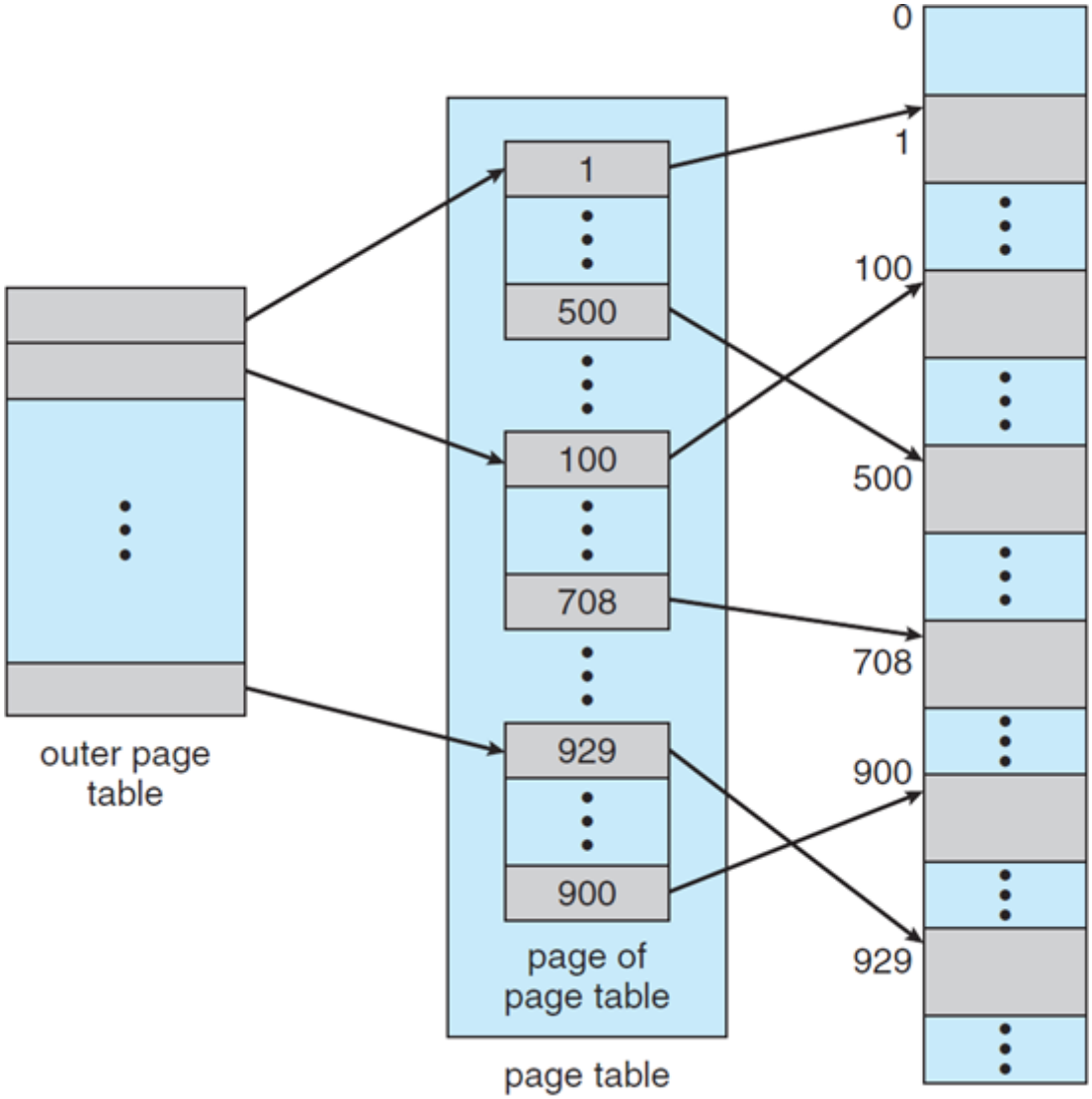
> 참고
>
> 32bit 주소 체계의 컴퓨터에서 인식할 수 있는 주소 공간은 2^32(4GB) 만큼이다. 따라서 RAM이 4GB보다 큰 경우는 인식하지 못하는 것!
>
> **이 말을 반대로 말하면 2^n 각각을 구분하기 위해서는 n bit가 필요하다는 말도 된다!**

현대의 컴퓨터는 address space 체계가 매우 크다.

32bit 주소 체계에서 인식할 수 있는 주소 공간은 4GB이다. 페이지의 크기를 4KB로 나누었다면 약 100만개의 페이지가 생기게 되고, 이를 위한 1M 개의 page table entry가 필요할 것이다. 그리고, 각 page entry는 4byte인데**(32bit 주소 체계니깐)**, 그러면 프로세스당 4MB의 page table의 필요하게 된다.

하지만, 대부분의 프로그램은 4GB의 주소 공간 중 지극히 일부분만 사용하므로 page table 공간이 심하게 낭비된다.

따라서 page table 자체를 또 하나의 page로 보는 방식을 사용했다.



Two-Level Paging 기법에서는 inner page table과 outer page table 두 가지의 테이블이 존재한다.

Two-Level Paging의 예

logical address (page size가 4K인 32bit 체계에서)의 구성은

- 20bit의 **page number** 와,
- 12bit의 **page offset** 으로 이루어져 있다.

  Why? 페이지의 크기는 4KB(4byte x 1K)이다. 즉, 2의 12제곱 바이트.

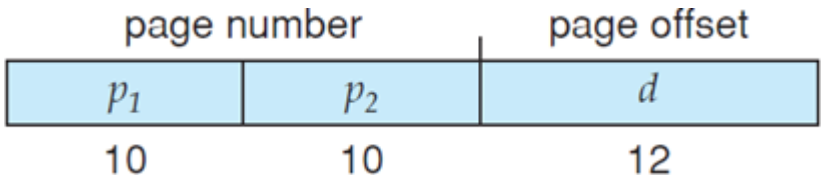
  메모리는 바이트 단위로 주소가 매겨지기 때문에, 페이지 안에 어디에 위치해있나**(몇 번째 떨어져있는 바이트인지)**를 확인하기 위해서 총 12bit가 필요한 것!

Two-Level Paging 기법에서는 page table 자체가 page로 구성되기 때문에, page number는 다음과 같이 나뉜다.

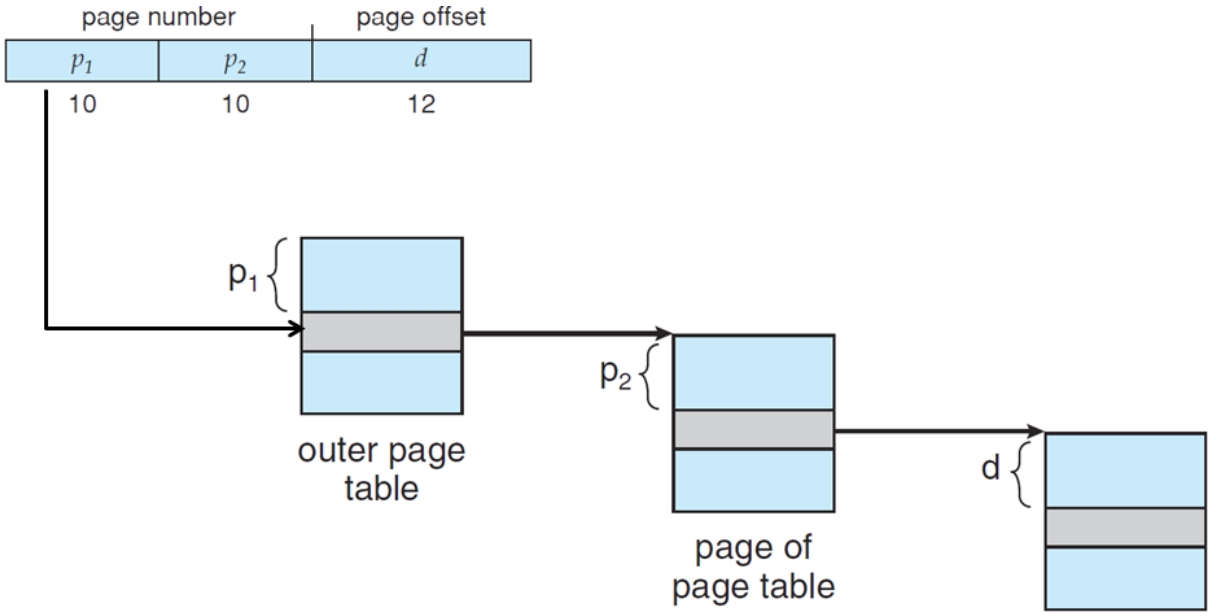
(각 **page table entry가 4B**이므로, entry는 총 1024(1K)개가 존재한다. 이를 구분하기 위해 10bit가 필요)

- 10bit의 **page number**
- 10bit의 **page offset**

따라서, logical address는 다음과 같다.



2단계 페이징에서의 주소 변환 scheme



그런데, 2단계 페이징 기법은 여전히 안쪽 페이지 테이블의 엔트리는 100만개가 필요하다.

오히려 바깥쪽 페이지 테이블이 하나 더 필요하기 때문에 1단계 페이징 기법보다 공간적으로나 시간적으로나 손해다.

- > 사용되지 않는 주소 공간에 대한 outer page table의 엔트리 값은 **NULL 값**을 갖는다.

Multilevel Paging and Performance

주소 공간이 더 커게 되면 다단계 페이지 테이블이 필요해진다.

각 단계의 페이지 테이블들은 메모리에 존재하기 때문에, logical address의 physical address 변환에 더 많은 메모리 액세스가 필요하다.

- > 이를 TLB를 통해 메모리 접근 시간을 줄일 수 있다.

4단계 페이지 테이블을 사용하는 경우

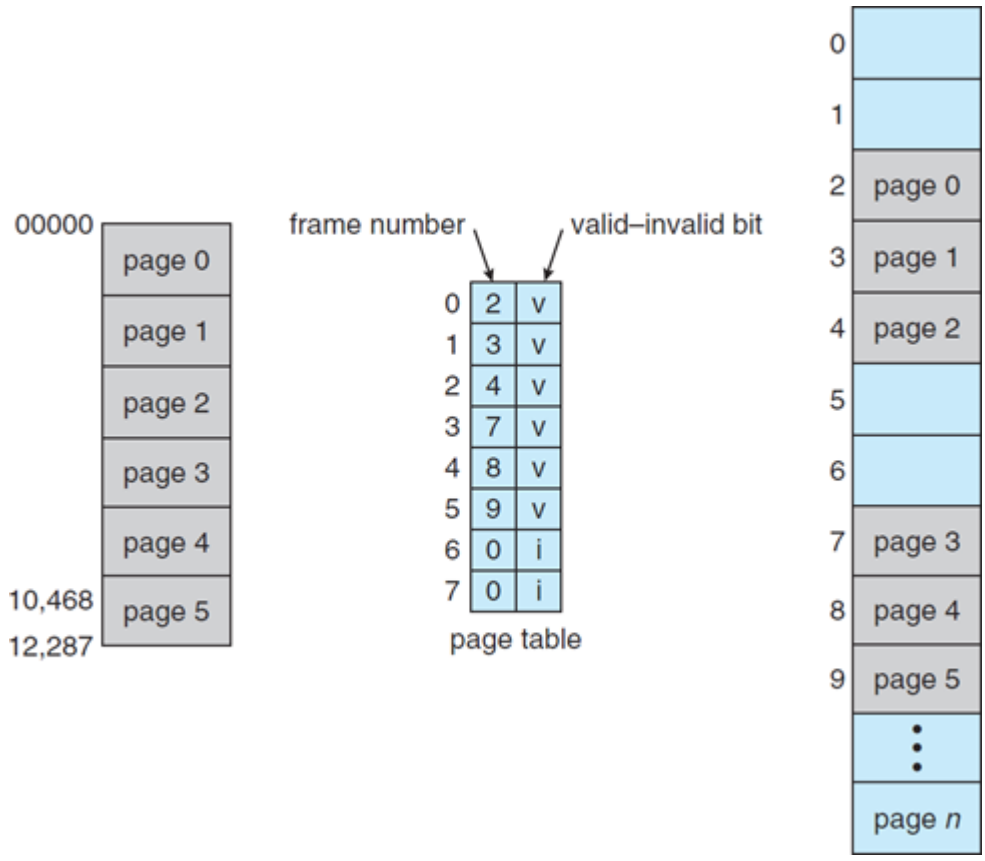
> Q. 메모리 접근 시간이 100ns, TLB 접근 시간이 20ns이고,
>
> TLB hit ratio가 98%라면 effective memory access time은 얼마일까?

0.98 * (100 + 20) * 0.02 * (100 * 4 + 100 + 20) = 128ns. 결과적으로 주소 변환을 위해서는 28ns 만 소요된다.

Page Protection

페이지 테이블에는 사실 frame number를 저장하는 비트말고 여러 비트가 있다.

- frame number
- valid-invalid bit
- protection bit



페이지 테이블은 인덱스를 통해 접근해야 하기 때문에 사용되지 않는 영역에 대해서도 엔트리가 생성되어야 한다.

위의 그림을 보면 6, 7번 엔트리는 사용하지 않기 때문에 frame number에 0이 들어가 있다. 하지만 이게 0번째 프레임을 가리키는건지 의미없는 값인지를 확인해주기 위해서 사용하지 않는 엔트리에는 invalid로 표시한다.

즉, valid로 표시되어 있단 것은 그 페이지가 **메모리에 올라와 있다**는 의미이다.

invalid로 표시되어 있으면 해당 주소의 **프레임에 유효한 내용이 없음**을 뜻한다.

- 프로세스가 그 주소 부분을 사용하지 않는 경우
- 해당 페이지가 메모리에 올라와 있지 않고 swap area에 있는 경우

Protection bit 는 페이지에 대한 접근(연산) 권한을 가진다. (read / write / read-only)

프로세스마다 각각의 페이지 테이블을 갖고 있기 때문에 프로세스간의 Protection을 의미하는 것이 아니다.

code 영역의 경우 사용자로부터 수정을 막기 위해 read-only 권한을 부여하고,

data 영역이나 stack 영역은 수정할 수 있기 때문에 read / write 권한을 부여하는 식이다.

PTLR (Page Table Length Register) - 페이지 테이블의 사이즈를 저장하는 레지스터이다. 페이지 테이블의 크기를 각각의 프로세스가 쓰는 만큼 줄여서 다른 프로세스의 페이지에

접근하는 것을 방지한다.

Inverted Page Table

페이지 테이블은 공간에 대한 오버헤드가 매우 크다.

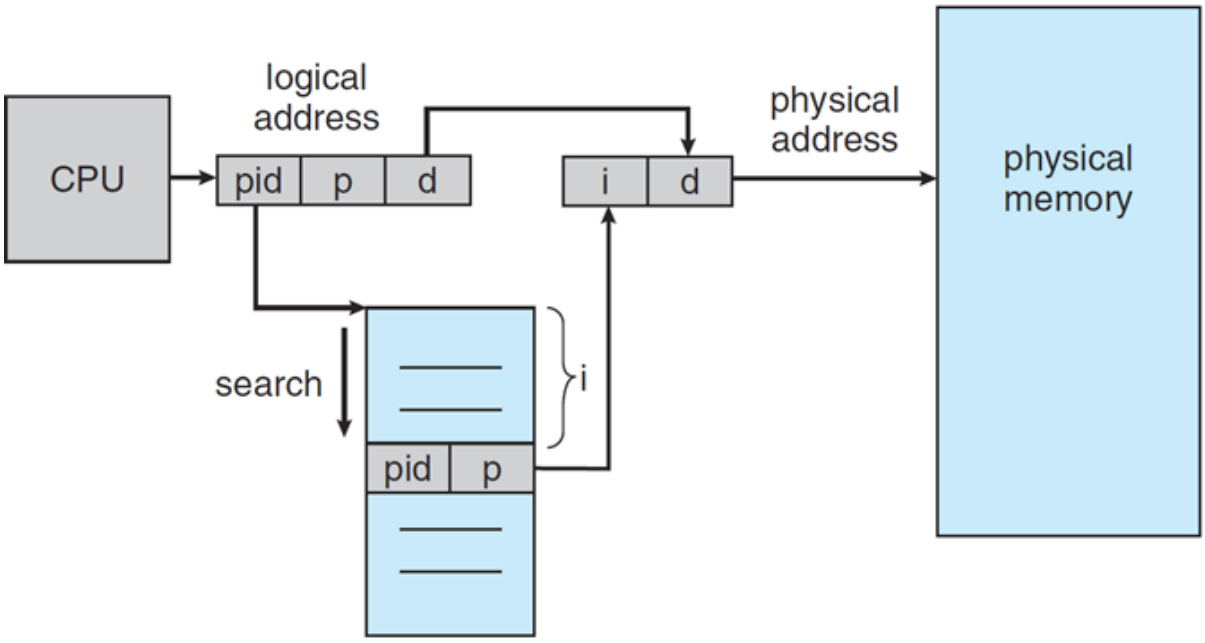
모든 프로세스 별로 logical address에 대응되는 모든 페이지에 대해 page table entry가 존재하고,

대응하는 페이지가 메모리에 있든 없든 페이지 테이블에는 엔트리로서 존재하기 때문이다.

기존의 페이지를 통한 주소 변환을 반대로 생각한 개념이 바로 Inverted page table 이다.

물리적 메모리의 frame 개수만큼을 담은 페이지 테이블을 갖는다. 따라서 **실제 메모리 크기 만큼의 페이지 테이블만 있으면 된다.**

logical address에 어느 프로세스가 사용하는 것인지 명시해주기 위해 pid 정보를 갖고 있어야 한다.



기존의 페이징 기법에서는 페이지 번호를 갖고 프레임 번호를 찾았는데,

Inverted page table 기법은 프레임 번호를 갖고 몇 번째 페이지에 있는지를 찾는다.

이 방식은 공간적인 오버헤드를 줄일 수 있는 대신, 시간적 오버헤드가 있다. 페이지 테이블 전체를 탐색해야 하기 때문이다. TLB 방식처럼 associative register를 사용해서 탐색해볼 수도 있겠다. 하지만 너무 비싸다.

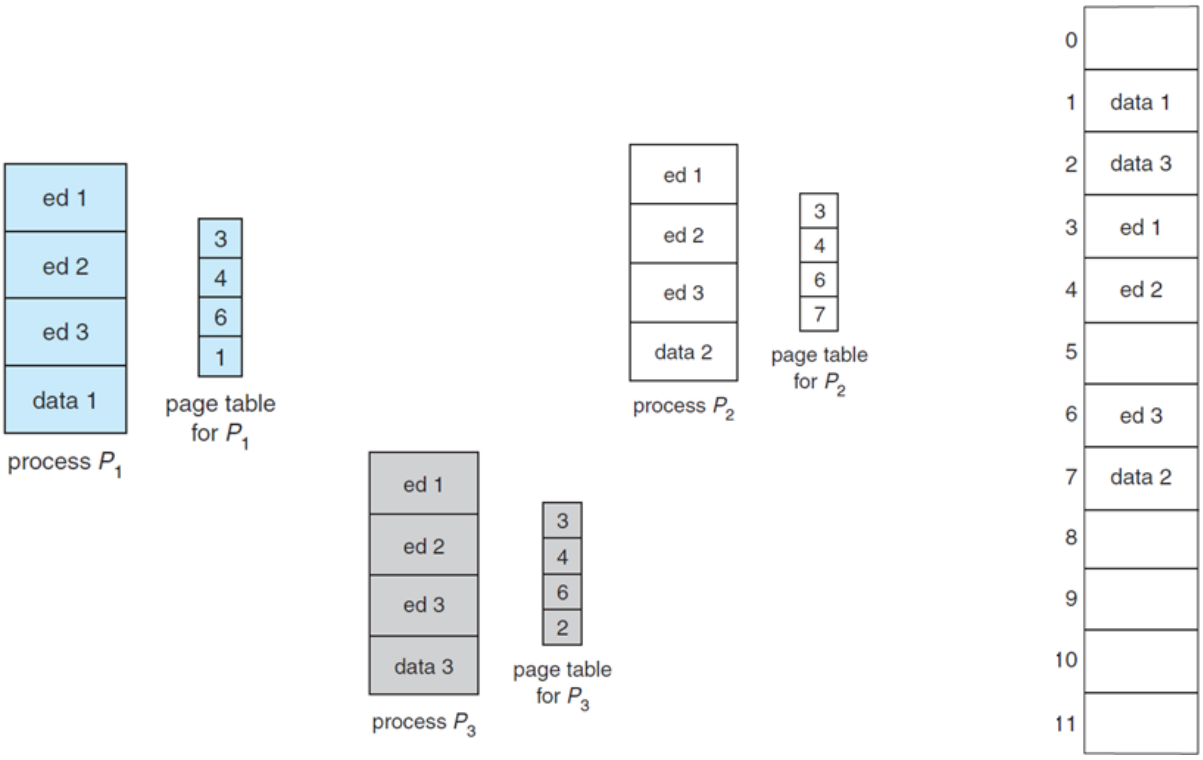
Shared Page

여러 개의 프로그램들이 똑같은 코드를 쓸 때, 코드가 들어있는 페이지의 중복을 없애보자.

동일한 내용이 들어있는 페이지(Shared code, Re-entrant code)를 모두 메모리에 올리는 것이 아니라 하나만 올리는 방식이다.

이러한 Shared-code가 있는 페이지는 반드시 read-only로 설정한다.

Shared-code는 모든 프로세스의 logical address space 에서 동일한 위치에 있어야 한다.



Segmentation

프로그램에 있는 여러 의미 단위를 각각 세그먼트로 구성한다.

크게는 프로그램 전체를 하나의 세그먼트로 정의할 수도 있고, 작게는 프로그램을 구성하는 함수 하나 하나를 세그먼트로 정의할 수도 있다.

일반적으로는 code, data, stack 부분이 하나씩의 세그먼트로 정의된다.

Segmentation Architecture

세그멘테이션에서의 주소 변환은 페이징 기법과 비슷한 면이 있다.

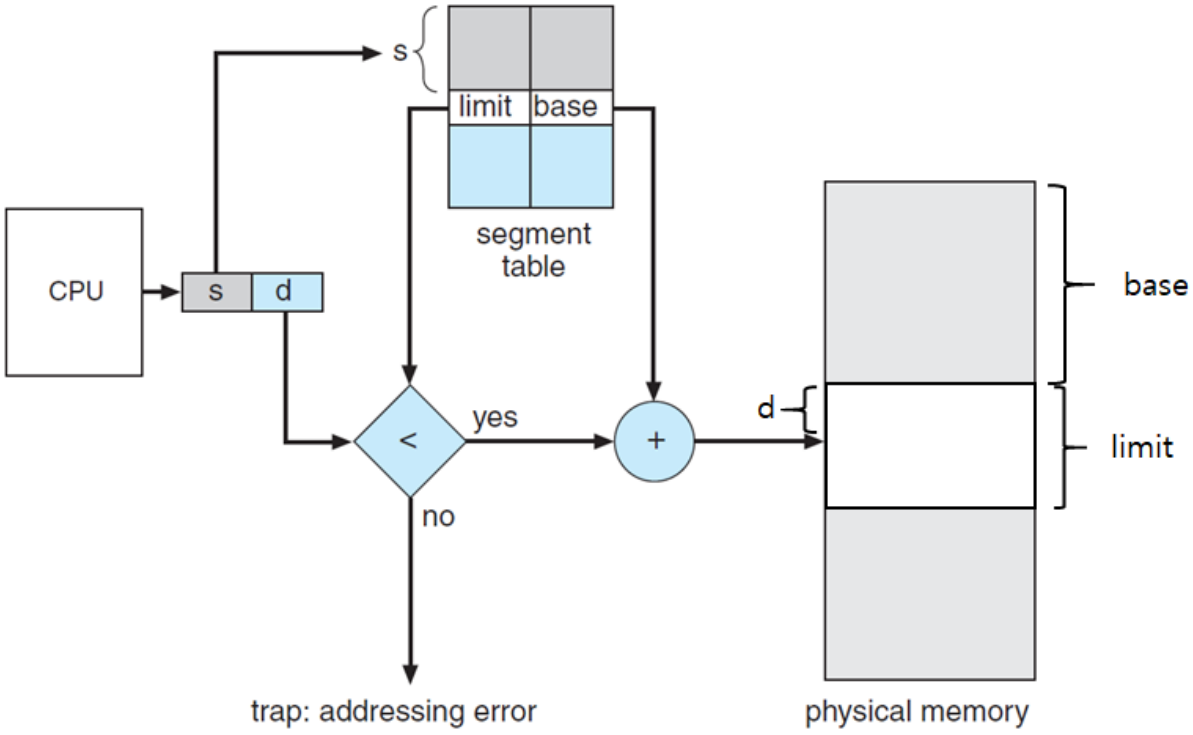
Logical address는 **<segment-number, offset>** 으로 구성된다. 여기서 offset은 세그먼트 안에서의 위치를 나타낸다. 세그먼트 별로 서로 다른 physical memory 위치에 올라가 있

기 때문에 세그먼트 별로 주소 변환을 해야 하므로 **Segment table**을 두고 있다.

주소 변환에 사용되는 기존의 **두 레지스터(base, limit)**는 다음과 같이 사용된다.

- Segment-table base register(STBR) - physical memory에서의 세그먼트 테이블의 위치
- Segment-table length register(STLR) - 프로그램이 사용하는 세그먼트의 수

Segmentation Hardware



trap: 원래 할당된 세그먼트의 범위를 벗어나서 액세스하는 경우이다. Segmentation fault ?

단점 - Allocation

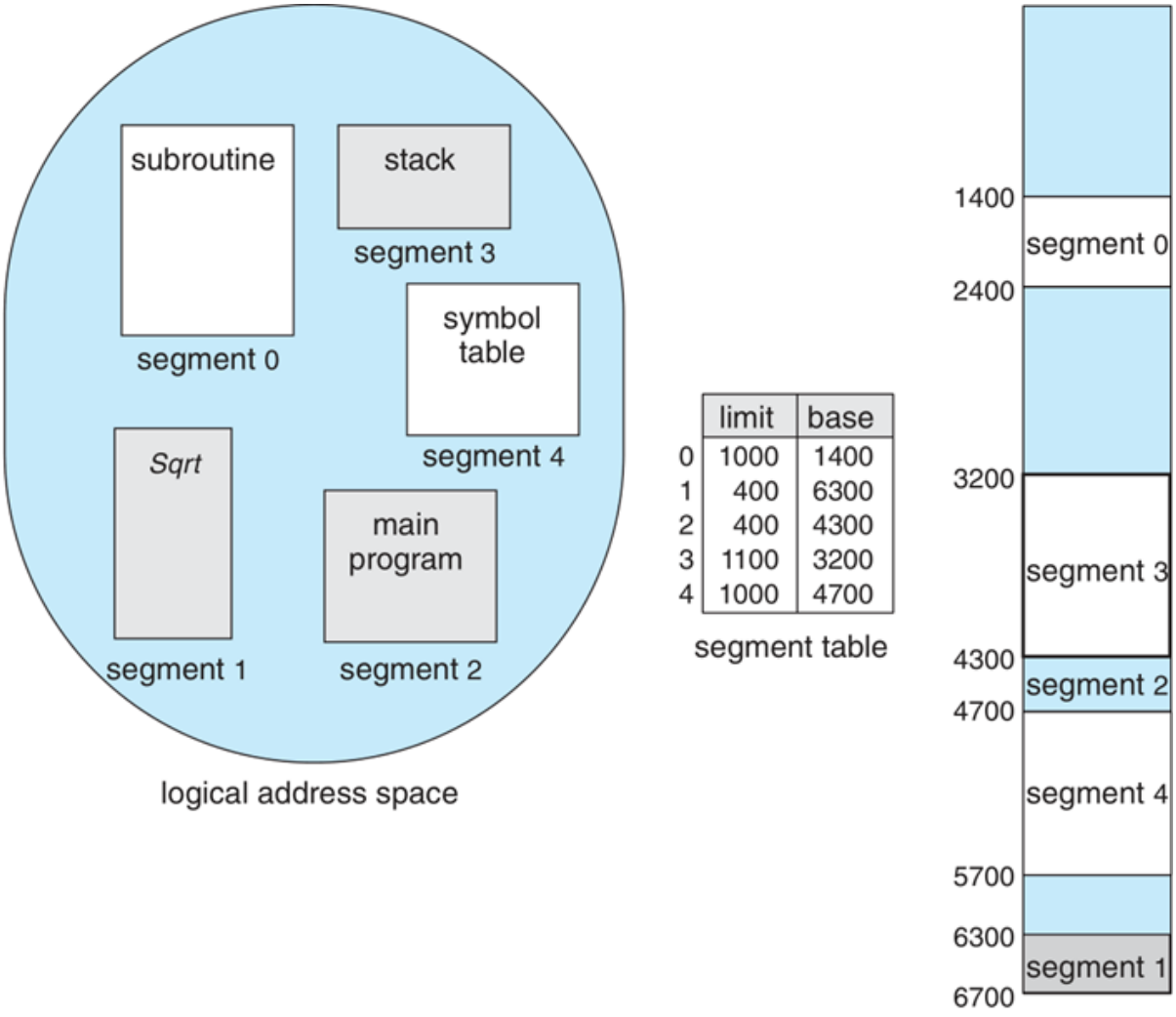
세그멘테이션 기법은 **External fragmentation**이 발생한다. 세그먼트의 길이가 모두 동일하지 않기 때문에 가변 분할 방식에서의 동일한 문제점들이 발생하는 것이다. 따라서 first bit / best fit 과 같은 방법을 사용한다.

장점

세그먼트는 의미 단위로 쪼개기 때문에 **공유**와 **보안**에 있어서 페이징 기법보다 훨씬 효과적이다.

- **Protection** - 각 세그먼트 별로 protection bit가 존재해서 의미 단위 별로 read/write와 같은 권한을 줄 수 있다.
- **Sharing** - 마찬가지로 의미 단위인 세그먼트를 공유하는 것이 훨씬 효과적이다.

Example of Segmentation



Paged Segmentation (Segmentation with Paging)

세그먼트 한 개가 여러 개의 페이지로 구성된다.

기존의 세그멘테이션 기법(pure segmentation)과 다르게, segment-table entry가 세그먼트의 base address를 갖고 있는 것이 아니라 세그먼트를 구성하는 page table의 base address를 갖고 있다.

즉, **선 세그멘테이션 후 페이징** 이다