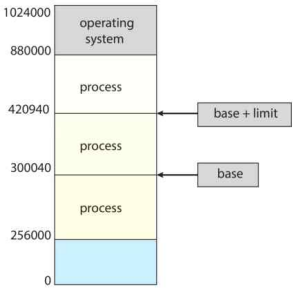
운영체제 9장
Main Memory(주 기억 장치 - 시스템 리소스)
-프로그램이 수행되려면 메모리를 할당받아 주 기억 장치에 로딩하여야 함
0. Background
메모리 기본 개념
1. Contiguous Memory Allocation
프로세스가 연속적으로 메모리 일정 부분에 빈 공간을 차지하는 것
2. Paging(프로세스가 일반적으로 상당히 커 연속적인 공간 부여 힘듦)
쪼개진 공간(일정한 크기) 여러 개를 할당하는 것
*Page와 Segment 차이점
Page는 일반적으로 크기가 작으나 Segment는 크기 다양함
module이나 성질이 다른 것을 다른 Segment 단위로 쪼개 프로세스에게 메모리 할당함
3. Structure of the Page Table
Page를 관리하기 위한 Page Table의 구조
4. Swapping
메모리 계층 구조에서 메모리 공간이 부족하여 더 넓은 공간 필요하며(HDD), 이로 인해 계층 내 아래 위로 Data가 이동하는 것
실제적으로 메모리 관리 어떻게 진행하고 있는지
5. Example: The Intel 32 and 64-bit Architectures
6. Example: ARMv8 Architecture(임베디드 모바일 전용 칩)

목표
-To provide a detailed description of various ways of organizing memory hardware
메모리 관리에서 어떤 Hardware가 지원되는지
-To discuss various memory-management techniques,
메모리 관리 위한 다양한 기법
-To provide a detailed description of the Intel Pentium, which supports both pure segmentation and segmentation with paging
Segment와 Paging을 융합하는 기법 알아보기

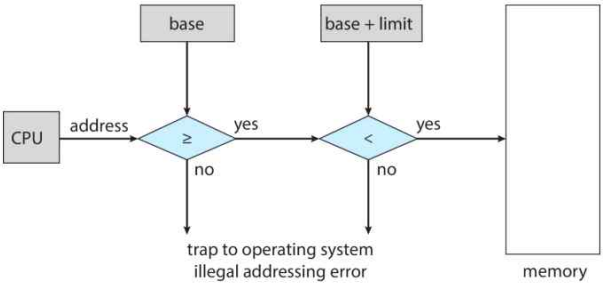
0. Background
메모리에 프로세스를 어떻게 할당할 것인가?
(1) CPU에서 메모리로 Data를 읽고 쓸 때에 주소가 필요함
- [addresses] and read requests
- [address + data] and write requests

(2) 메모리 계층 구조
-Register, Cache, Main memory, Swap 공간(보통, HDD or SSD)
-Protection(메모리 관리에서 가지고 있는 기능)
메모리 공간을 여러 프로세스가 영역 구분하여 자신이 할당받은 부분만 사용

#Contiguous Memory의 Protection 예시



OS : 연속적인 공간 사용
각 Process : 연속적인 공간 할당받아 사용
#이들은 register에 저장
(이것 또한 메모리에 저장한다면 memory access마다 다시 memory에 접근하여 확인해야 하므로 성능에 문제)
-base 시작 주소
-base + limit(자기 proc의 크기)
✓ base 아래 영역 사용하지 않는지, base+limit 위 영역 사용하지 않는지 Check



앞선 조건 만족하지 않으면, Trap이라는 OS의 error 발생함
(Base와 Base limit 설정하는 instruction은 privileged함-kernel, system 모드에 있는)

(3) Address Binding
실제로 프로그래밍 작성->소스코드 컴파일->오브젝트 파일->실행파일 등의 일련 과정 중 언제 Address가 정해지는가? 하는 것

Source code 작성할 때, 내가 선언하는 table, variable이 메모리 어디에 위치할지 모름
✓ 이 Address값은 프로그램이 수행되어 메모리에 로딩될 때 결정됨

Source code - 보통 symbol로 결정
Compiled code - relocatable 주소(재배치 가능함, 메모리 올라갈 때마다 변동O)에 바인딩
▶ 컴파일 코드 내 상대적 위치만 결정된 상태(ex. 이 모듈 내 14번째 byte를 access하겠다)
Linker or loader - 컴파일이 된 후 로더에 의해 실제 주소가 결정됨

(4) Binding of Instructions and Data to Memory
-Compile time: 일반적으로, 우리가 절대 주소를 알 수가 없음(상대주소만 결정)

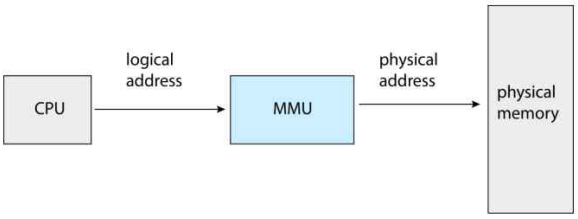
*일부 특수한 사례(Bootloader 등)만 절대주소 결정됨
-Load time: 프로그램이 메모리에 로드되는 단계를 나타내며, relocatable code 생성 (메모리에 로딩되어 수행 시, 실제 주소가 결정되는데 이 code로 작성되어 있다면, 상대주소+모듈 로딩 위치=실제 주소 산출에 도움)
-Execution time: 컴퓨터 프로그램/코드의 명령이 실행되는 단계 나타냄. 실행 시, 런타임 라이브러리 사용됨 (그 프로그램이 호출되는 순간)
*메모리 낭비 줄이고자, 모든 code가 로딩되는 것이 아니라 일부 code만 올리기 때문에
(특정 함수가 호출되었을 때 그 proc의 부분을 load한다, DLLs)
✓ 세 가지 작업이 모두 CPU에서 수행됨
✓ 주요 차이점

Compile Time	**Load Time**	**Execution Time**
Code is translated into machine readable format	Program code is loaded into memory	Program code is executed
Completed with a compiler	Completed with a loader	Completed with run-time libraries

(5) Logical vs. Physical Address Space
-Logical address(=virtual address): CPU에서 본, proc에서 본 address를 말함
proc마다 충분한 연속적인 공간이 확보되어 있다고 봄(상대적 or 가상의 주소)

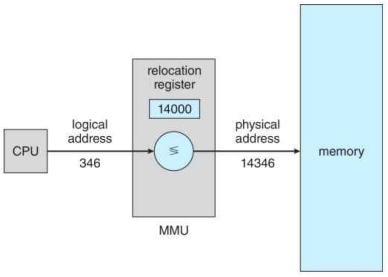
-Physical address: 실제 physical memory 상 주소
시스템에서 관리하는 H/W memory(주 기억 장치)의 주소

(6) Memory-Management Unit (MMU, 일반적으로 CPU 내 포함되어 있음)



CPU 입장에서 Logical address로 보고(프로그램 짤 때 항상 0부터 N줄까지), 이것이 MMU에 의해 address mapping이 일어나, physical address로 변환됨

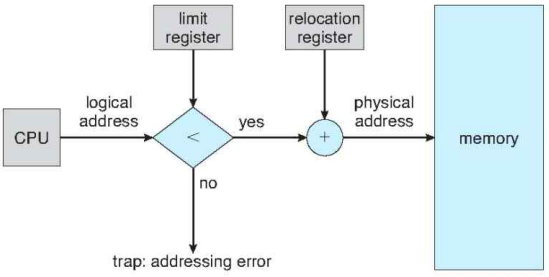
-Relocation register
memory-management를 위해서 필요한 기본(base) register이며, 컴파일 시에는 자신이 어느 주소에 실제로 할당되어 있는지 알 수 없음
logical + relocation register 값 -> physical 로 주소변환이 일어남



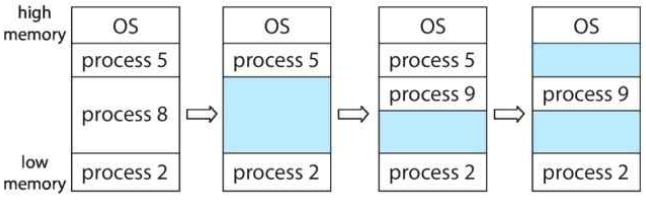
-Dynamic Loading
프로그램 시작될 때, 모든 코드가 한꺼번에 메모리에 로딩되는 것이 아니라, 필요한 부분(프로그램이 수행되다가 그 함수나 모듈이 호출될 시)이 유연하게 로딩되는 것

-Dynamic Linking
Static linking
 실행 파일을 만들 때 한꺼번에 link되는 것
Dynamic linking
 그 부분이 호출될 때, 그때서 linking이 되고 loading이 되는 것
잘 호출되지 않는 error 처리 루틴, 공유 라이브러리 등에 대해 D.Linking, Loading 사용하여 메모리의 효용성 ↑

1. Contiguous Memory Allocation (실질적으로 메모리 공간 어떻게 관리하는지)
(1) 몇 가지 개념
-partition : 한 proc가 차지하고 있는 하나의 조각같은 공간
-Base register : proc의 시작 주소를 저장하는 공간
-Limit register : proc가 차지하고 있는 메모리의 크기 저장하는 공간
(OS는 메모리의 빈 공간을 관리함 - 어느 부분에 proc가 요구하는 빈 공간을 할당할까?)
-주소 공간이 dynamically함(해당 공간 사용하다가 넘겨주거나, prog 수행 위해 메모리 할당받을 때 아까와 같은 주소 공간 할당받는다는 단언 X)



(2) Variable Partition(<->static parition: equal, unequal OS 초기화 시 사전 분할 가능)
메모리 공간에 proc가 사용하는 경계(partition)을 설정해야 하는데, proc마다 크기가 다르고 요청하는 것이 달라 그때마다 parition을 설정한다는 것(메모리 낭비 줄일 수 있음)



#생각해야 하는 고려점(메모리 빈 공간 존재 시, proc 할당 요청을 어떻게 만족시킬 것인가?)
First-fit : 위에서부터(낮은 주소) 찾아가며 proc 요청 만족하는 빈 공간 찾아 할당
Best-fit : 전체 memory 공간을 찾아 proc 요청에 (크거나 같은 공간 중)가장 잘 맞는 공간 할당[할당 시에 공간 낭비 줄이나 속도 ↓]
Worst-fit : 전체 memory 공간을 찾아 가장 큰 공간을 할당 [공간 낭비 ↑, 속도 ↓]

(3) Fragmemtation
-External Fragmentation: proc가 차지하는 partition 사이, proc가 사용 못하는 작은 공간
-Internal Fragmentation: static parition 경우, proc 할당 시, 해당 partition 내 남은 사용하지 못하는 공간

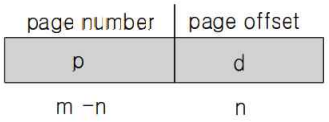
-External Fragmentation 해결책 : Compaction
proc 할당한 partition 자체를 partition 사이 공간만큼 위로 움직여, 빈 공간 없앰

#앞서 얘기한 것은 Contiguous Memory 할당 개념!
2. Paging
메모리를 작은 Page로 쪼개어 관리하는 것(연속적 공간 확보 필요 X)
(일반적으로 512 bytes 정도, OS같은 것을 위한 공간은 16M bytes만큼 크게 쪼개기도)

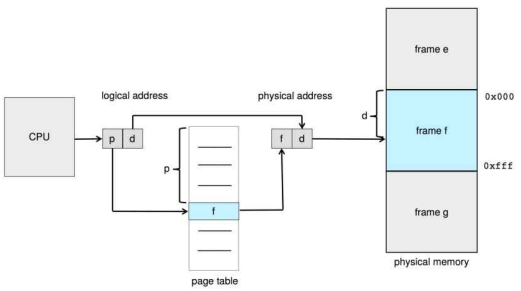
-고안 동기
physical memory 공간->noncontiguous 가능성(할당 가능 연속적 공간 찾기가 더 어려움)
external fragmentation도 빈번히 발생함
✓ Paging 기법 고안 - 현존하는 OS들은 대부분 Paging 기법을 활용하고 있음
ex. page 당 1K 크기일 때, 10K 프로그램 수행을 위해서 10 page 할당 가능

-Paging 단점
Internal Fragmentation 문제 존재

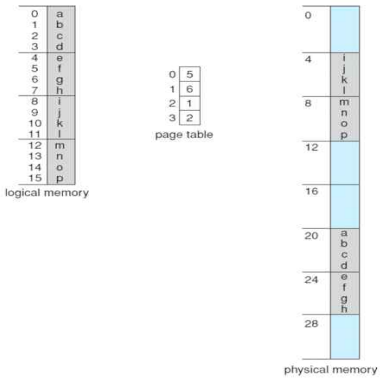
(1) Address Translation Scheme
Paging 기법 내 addresss 변환(page 자체는 logical하게는 연속적으로 존재)
> Page table(proc마다 page table이 독립적으로 존재)에서 address 변환이 일어나, physical memory의 어디를 할당받을 것인가 결정



-Page number (p): 몇 번째 page냐 하는 것(logical한 순서)
-Page offset (d): 그 page 내 상대 주소
*Page == Frame [Page: logical한 개념 <-> Frame: physical한 개념]



#Paging Example



(2) Paging -- Calculating internal fragmentation

Paging의 경우에 발생하는 internal fragmentation 계산 방법

-Page size, Process size 이용

Process size / Page size = Page 개수 + 남은 byte(하나의 page 할당)

Page size - 남은 byte = internal fragmentation

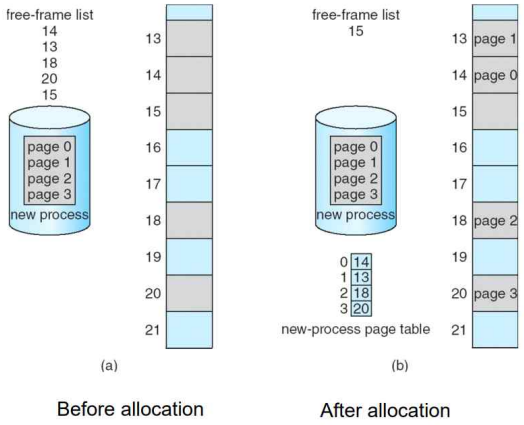
> internal fragmentation 줄이기 위해 solaris의 경우 두 page 규격을 지원하여 할당함

(+ memory 크기 ↑함에 따라 page table 크기도 ↑)

(3) Free Frames

Process에게 Memory를 할당하고 남아있는 Free한 공간

free-frame list로 이들을 관리하며, list 위에서부터 하나씩 새로운 proc에게 할당함



(4) Implementation of Page Table

Page table의 경우 원래 Memory에 존재하는데, Memory에 access하기 위해서 Page table에 memory mapping을 거쳐야 함. >> Memory에 2번 access 하게 됨

두 가지 register에 올려서 사용

Page-table base register (PTBR) : page table 가리킴

Page-table length register (PTLR) : page table의 크기 가리킴

> Register를 두어도 page table이 memory에 있어 두 번 access하게 됨(속도 ↓)

✓ TLBs or associative memory - Page table을 cache하여 속도 향상

(5) Translation Look-Aside Buffer(TLBs)
일종의 Cache - context switching할 때 잘 보관해야 함
proc마다 가지고 있는 것이 아닌, System이 공통으로 사용해야 함
✓ 어떤 proc가 사용하고 있는지 판단(TLB table 내 구별 가능 도구 필요)
✓ 현재 수행하는 proc가 이 TLB table을 독점하도록

따라서, address translation을 위해 page table 찾기 전에 먼저 TLB table을 찾아 확인하고 cache에 있으면 그것을 사용, 아니면 page table 확인함

(6) Hardware
page table에는 모든 page에 대해 연속적으로 mapping되는 frame 값이 존재하지만,
Cache(TLB table)에는 일부만이 저장되어 (mapping되는 frame 정보 + 해당 page 번호)를 함께 저장

(7) Effective Access Time
-Hit ratio : TLB 내에서 page number가 발견될 확률
-Effective Access Time (EAT) 계산

(8) Memory Protection
Valid-invalid : 자기 영역 바깥에 있는 것을 access하면 안된다는 것
Page 구분 : page마다 read-only, read-write인지 구분하는 것

(9) Share Pages
Page table을 이용하면서 proc간 shared code, data 서로 공유할 수 있는 좋은 구조 형성
ex. proc간 라이브러리 공유
private code, data : proc마다 혼자 사용하는 것

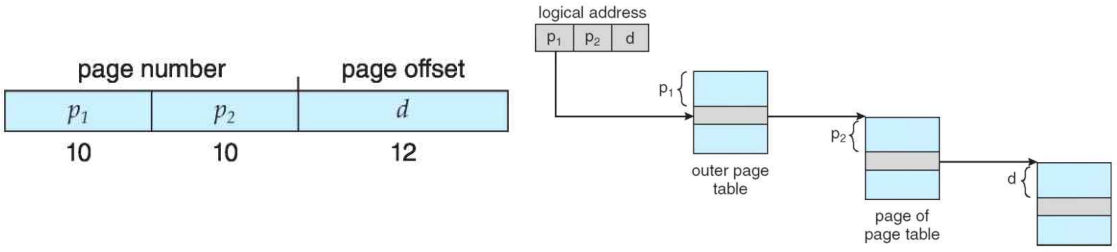
#page table 그대로 사용하면 문제점 존재 ex. page table의 크기가 너무 커짐
3. Structure of the Page Table
32-bit 컴퓨터에서 page size가 2^12일 때, page table은 2^20개가 나옴
>page table entry마다 4byte라면, physical memory에 전체 page table은 4MB차지함
(다 쓰지도 않는 page table을 위해 physical memory 공간 낭비)

-Page table 크기 및 낭비 해결책
🗐 Hierarchical Paging
🗐 Hashed Page Tables
🗐 Inverted Page Tables

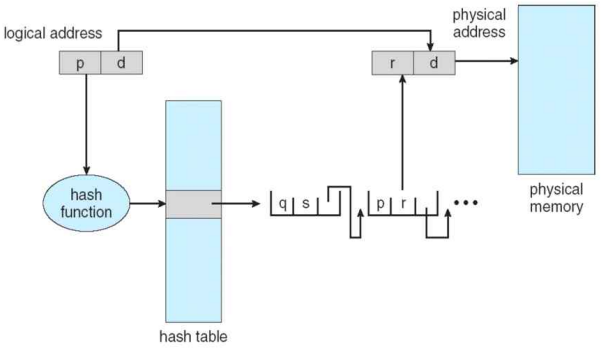
(1) Hierarchical Paging
계층 구조를 이용하는 것, memory를 page로 나누어 관리함(page table 이용)
▶ page table이 커서, 그 자체를 다시 page로 쪼개어 관리하는 것

(그 쪼갠 것을 다시 관리하는, page table의 page table을 outer page table이라고 함)
=two-level page table

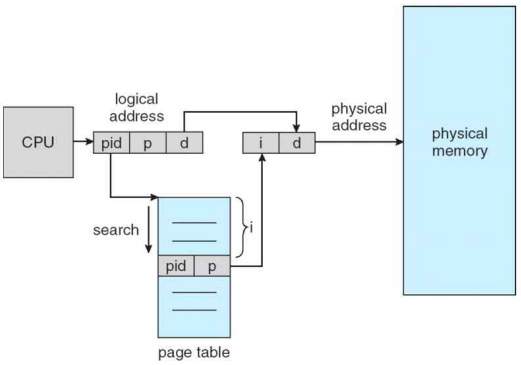
#Two-Level Paging 예제(p가 2개 있음)



(2) Hashed Page Tables(Hash table 이용)
Hash도 일종의 index이고, page에서의 주소를 가지고 hashing 진행해 빠르게 검색 가능
clustered page tables : 한꺼번에 모여있는 page들을 가르킬 수 있음
(뒤에 나올 넓은 공간 연속해서 쓰고 싶은 OS나 자료구조 위해 MB 단위의 page table도 크고 작은 page 섞어서 사용할 수 있는 기법과 유사)
> hash function(x) = x % 1000 일 때, 1033, 2033, ..10033 등 충돌 가능
✓ linked list로 해당 hash 값을 가진 것들 모두 저장(그들 사이 구분 값 같이 저장)



(3) Inverted Page Tables



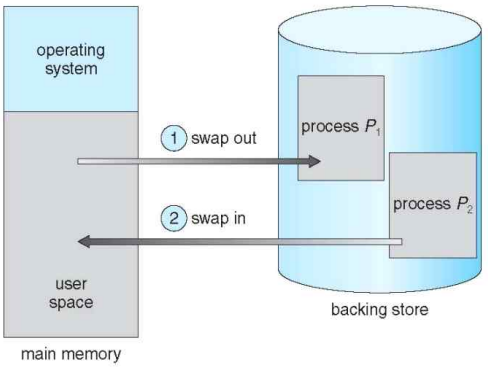
logical 주소 p를 page table에서 찾아, 그 때의 index i가 frame number가 됨
Inverted page table의 entry는 physical frame 수만큼 존재하면 됨(index i = frame 수)
-시스템이 table을 공유하므로, 어느 proc가 p를 사용하는지 구분하기 위해 pid 포함

-특징
일반적인 page table은 proc마다 존재하는데 반하여, Inverted page table은 한 시스템마다 하나씩 존재하며, 그 size는 physical memory의 Frame의 개수만큼만 있음
page table 크기 줄이는 효과 존재

-단점
p를 찾기 위한 시간이 많이 소요됨

Oracle SPARC Solaris 예시
> page 기법이 기본이지만, 그대로가 아니라 table size 줄이기 위해 hashing 활용
> hash table 사용하면서 contiguous area(연속적인 커다란 공간) 설정해 효율적 사용 가능
> TLB (page table을 캐시한 것) 개념 이용

4. Swapping
(1) Swap 공간 등장 배경
Memory를 여러 proc가 공유하고 prog size가 증가하면서 더 넓은 공간 확보 필요
✓ Memory 공간이 부족할 때만, 일부 proc가 HDD swap 공간으로 이동하여 공간 확보
-Backing store: Memory 공간이 좁아 더 넓은 공간 확보 위해 HDD의 Swap Area
-Roll out, roll in: Swap in, out과 동일한 개념
-거의 모든 OS에서 Swapping 공간 사용하고 있음(ex. Unix, Linux, and Windows)



(2) Context Switch Time including Swapping
-Context switch time: Swap out된 proc에 대해서는 기존 switching 시간+swap in시간
✓ 기존 Context Switching에서 proc가 CPU 할당 및 이동 시간, Swap 시간까지 고려 필요

-I/O operation(메모리 사용) 등은 swapping의 제한 요소로 작용
커널에서 관리하는 buffer 통해 I/O request, response data 저장 등을 진행함
I/O 진행 중 Swap out당하면, 본인 memory와 요청한 I/O 공간 memory 흐트러질 수 있음

(3) Swapping on Mobile Systems
-특징
경량화, 저전력 위해 Flash memory 사용(공간 ↓, 제한된 write cycle 횟수)
wireless 통신 이용(mobile communication)

-iOS, Android 같은 모바일 OS에서의 memory 관리
주 기억 장치 공간이 부족할 때, Swap 수행 X고, 수행하고 있는 prog을 종료함
(Swap 수행 -> Flash memory의 write cycle에 부담됨)

#iOS
a. application이 알아서 memory 공간 반납하고 가급적 read-only한 data 종료를 고려함
b. proc를 termination 시켜서 memory 내 지움

#Android
a. 돌던 prog을 종료 시, prog 수행하던 기본적 상태는 flash memory에 저장
b. proc를 termination 시켜서 memory 내 지움

5. Example: The Intel 32 and 64-bit Architectures
H/W적으로 memory 관리를 지원해야 함
(1) Example: The Intel IA-32 Architecture
거의 모든 OS가 **segmentation**와 **segmentation with paging(segment->page로 쪼갬)** 지원
-segmentation
성격으로 분류하는 것 ex. text, data. library
page가 크기로 쪼개는 것과 달리, 모듈, 성질별로 쪼개며 비교적 page보다 크다

-sement table: segment 관리 위한 것
종류 : local descriptor table (LDT), global descriptor table (GDT)

-segmentation vs. paging

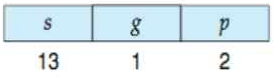
S.NO	페이징	분할
1.	페이징에서 프로그램은 고정 또는 마운트 크기 페이지로 나뉩니다.	분할에서 프로그램은 가변 크기 섹션으로 나뉩니다.
2.	페이징 운영 체제에 대한 책임이 있습니다.	세분화 컴파일러는 책임이 있습니다.
삼.	페이지 크기는 하드웨어에 의해 결정됩니다.	여기서 섹션 크기는 사용자가 지정합니다.
4.	분할에 비해 빠릅니다.	세분화가 느립니다.
5.	페이징으로 인해 내부 조각화가 발생할 수 있습니다.	세분화는 외부 단편화를 초래할 수 있습니다.
6.	페이징에서 논리 주소는 페이지 번호와 페이지 오프셋으로 나뉩니다.	여기서 논리 주소는 섹션 번호와 섹션 오프셋으로 나뉩니다.
7.	페이징은 모든 페이지의 기본 주소를 포함하는 페이지 테이블로 구성됩니다.	분할은 또한 세그먼트 번호와 세그먼트 오프셋을 포함하는 세그먼트 테이블로 구성됩니다.
8.	페이지 테이블은 페이지 데이터를 유지하기 위해 사용됩니다.	섹션 테이블은 섹션 데이터를 유지합니다.
9.	페이징에서 운영 체제는 사용 가능한 프레임 목록을 유지해야 합니다.	분할에서 운영 체제는 주 메모리의 구멍 목록을 유지 관리합니다.
10.	페이징은 사용자에게 보이지 않습니다.	분할은 사용자에게 표시됩니다.
11.	페이징에서 프로세서는 절대 주소를 계산하기 위해 페이지 번호와 오프셋이 필요합니다.	세그먼트화에서 프로세서는 세그먼트 번호와 오프셋을 사용하여 전체 주소를 계산합니다.
12.	프로세스 간 절차 공유를 허용하기 어렵습니다.	프로세스 간 절차 공유를 용이하게 합니다.
13	페이징에서 프로그래머는 데이터 구조를 효율적으로 처리할 수 없습니다.	데이터 구조를 효율적으로 처리할 수 있습니다.
14.	이 보호는 적용하기 어렵습니다.	세분화된 보호를 위해 적용하기 쉽습니다.
15.	페이지 크기는 항상 프레임 크기와 같아야 합니다.	세그먼트 크기에는 제한이 없습니다.
16.	페이지는 정보의 물리적 단위라고 합니다.	세그먼트는 정보의 논리적 단위라고 합니다.
17.	페이징은 시스템 효율성을 떨어뜨립니다.	세분화는 보다 효율적인 시스템을 만듭니다.

-구성

Segment table에서 변환해야 하는 segment number

LDT/GDT 구분하는 g

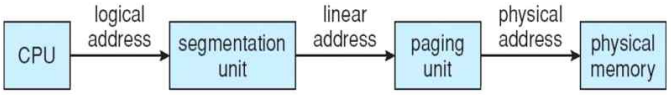
production(read, write 구분 + 어떤 operation 가능한지) p



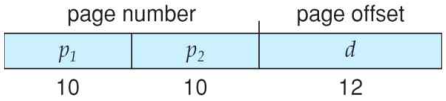
Linear address given to paging unit(size 4KB or 4MB)

#Address Translation 잘 보여주는 예시

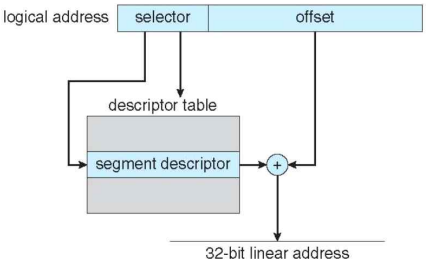
segment를 기반으로 하고 있지만, segment 또한 page로 관리



segment table이 logical -> linear로 바꾸며, linear address는 page 기법으로 관리



-Intel IA-32 Segmentation(logical -> linear)

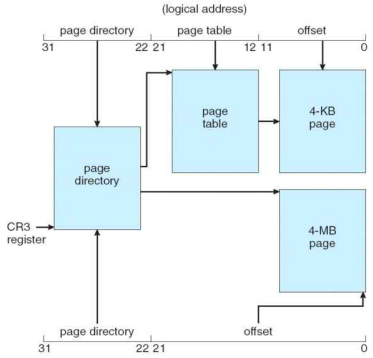


segment table(=descriptor table)

segment number(=selector)

segment descriptor: 시작 주소

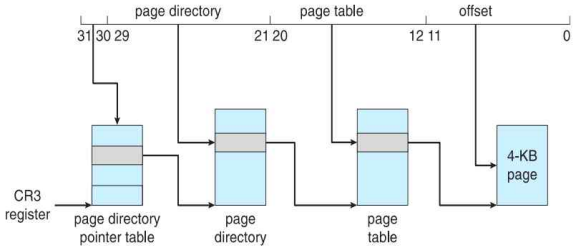
-Intel IA-32 Paging Architecture(linear -> physical, paging 기법)



page table(=page directory)

CR3 register : 프로세스마다 가지고 있는 page directory의 시작 주소(인텔 32bit의 특징)

-Intel IA-32 Page Address Extensions



32bit-address로 가리킬 수 있는 영역은 4GB 정도의 memory 공간
But, segment descriptor의 bit 수를 늘림으로써 address를 넓히는 효과 가능
(4bit 늘려 4GB의 16배인 64GB까지 확장해 사용 가능)

(2) Intel x86-64
address 영역 충분하여, virtual address 48bit와 physical address 52bit만 사용
-page size를 필요에 따라 다양하게 가져감(4KB, 2MB, 1GB)

6. Example: ARMv8 Architecture
모바일에서 많이 사용하며, IOS나 Android과 잘 궁합이 맞는 구조
저전력, page 기법, section, two-level page 계층, TLBs