



# 11. Multi-Level Page Tables

→ page table 작게 관리하기 위해서 제안된 방법  
(효율적이지!)

## ▼ Paging ⇒ page table 크기 계산해보자

### ▼ Linear page table (하나의 큰 테이블로 존재)

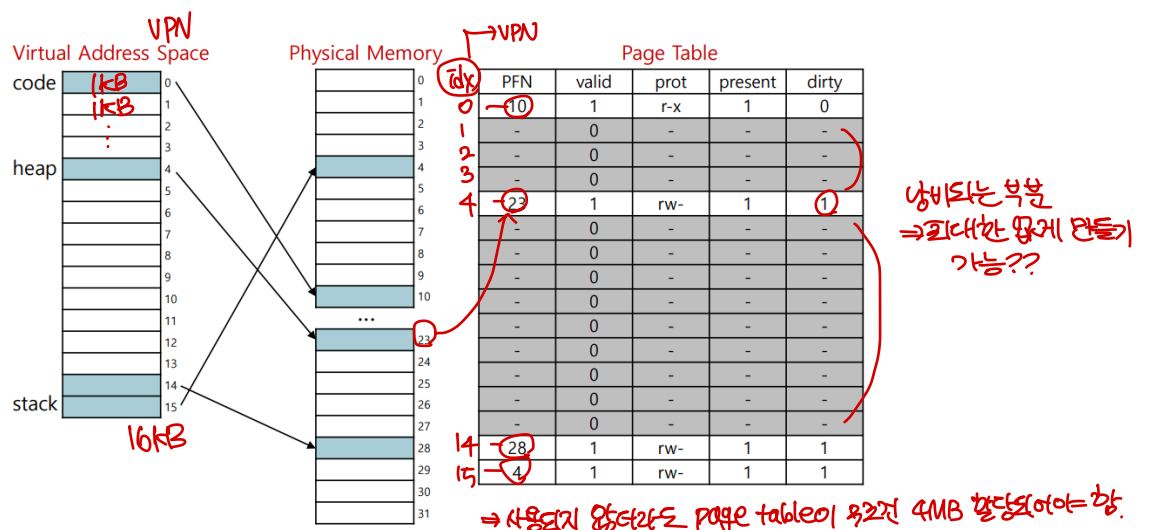
- page table : 너무 커서 메모리 소모량이 많음
  - abstraction을 적용하기 위한 memory 낭비도 큼 → cpu에 있기 어려움
- example

- 32-bit address space with 4KB pages  
↓  
2<sup>12</sup>B → offset : (2 bit)
  - 4byte PTE ★
  - 4MB per page table
- VPN : 20 bit → page table 2<sup>20</sup>  
page table 크기 : 2<sup>20</sup> × 4B = 4MB  
→ 한 page 크기

⇒ 메모리에 계속 있어야 하는데.. 더 작게 관리할 순 없을까?

+) 작게 유지해서 생기는 좋지 않은 영향도 있지 않을까?

- page table 어떻게 작게 만들까?
  - 새로운 data structure는 없을까?
  - 새로운 걸 사용하게 되면 어떤 비효율성 발생?



example : 16KB address space with 1KB pages

## ① ▼ simple approach

- bigger pages  $\Rightarrow$  해결하기 위해 page size만 키워보자!!
- page size 커짐  $\rightarrow$  page table entry의 정보는 작아짐  $\rightarrow$  cache, heap (page가 클수록  $\Rightarrow$  entry도 작아)

◦ example

- 32-bit address space with 16KB pages  $\rightarrow 2^4 \times 2^{10} = 2^{14}$  (4KB  $\rightarrow$  16KB (4배))
- 1MB per page table (4MB  $\rightarrow$  1MB (4배))

◦ Big page  $\rightarrow$  각 page 내에 낭비되는 놈 존재

- internal fragmentation page 내에서 사용하는 공간이 작음.  $\rightarrow$  page size를 커질수록 문제 심각

- process가 필요한 양보다 더 큰 메모리가 할당되어 공간 낭비 발생 (page 내에서의 나머지 부분이 커짐...)
- process : 실제로 사용하지 않는 메모리 영역 가지고 있게 됨

( $\leftrightarrow$  external fragmentation : 메모리는 남는데 너무 쪼개져서 사용 불가능한 경우)

$\Rightarrow$  대부분의 system : 보통 상대적으로 작은 page size 사용

- 4KB in x86, 8KB in SPARCv9

## ② ▼ Hybrid approach $\Rightarrow$ paging, segment 둘다 사용해보기!

• paging and segments

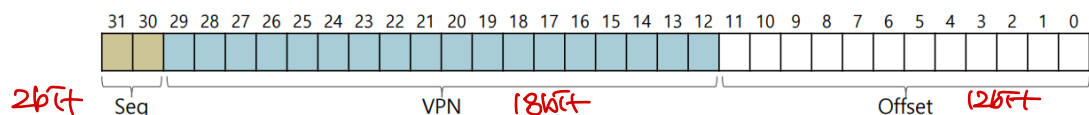
- 각 segment마다 하나의 page table
- 3개의 base/bound pair  $\rightarrow$  segment를 기본적으로 3개로 나눌 것다는 뜻
- base register : segment마다 page table의 물리적인 주소를 가지고 있음
- segment 스스로 point x
- bounds register : segment 내부 최대 valid page의 값을 가지고 있음

★ linear page table에 비해 memory saving 효과가 큼  $\rightarrow$  나뉘는 가장 좋은 분기 가져야 함

- stack, heap 사이에 할당되지 않은 page : 더 이상 page table 공간 차지 x

$\rightarrow$  관리 안 해도 됨!

◦ example



$SN = (VirtualAddress \& SEG\_MASK) \gg SN\_SHIFT$   
 $VPN = (VirtualAddress \& VPN\_MASK) \gg VPN\_SHIFT$   
 $AddressOfPTE = Base[SN] + (VPN * sizeof(PTE))$

entry의 시작주소

이런 segment를 위한 page table의 base address?

- 32-bit address space with 4KB pages

offset : 12bit  
vpu : 18bit  
seg : 2bit

- 4 segments

- 00 : Unused → 해당 example에서 CPU가 이렇게 정의
- 01 : Code
- 10 : Heap
- 11 : Stack

### 단점 ★

- ① 크지만 부족하게 사용되는 heap : 낭비되는 page가 많음

ex) malloc 잔뜩 할 → 맨 마지막 free X.  
나머지 free O

- heap에 대해서는 linear한 page table로 관리해야 함
- but, page table이 가변적으로 변해버림
- free → waste 발생 / allocate → 더 큰 공간 필요

⇒ 그만큼 linear한 entry들을  
page table 관리해야 함.  
⇒ 낭비

⇒ memory copy overhead가 발생할 수 있음

- ② external fragmentation → segment 크기 ↑ page table ↑ → memory

공간 다시 찾아야 함.

- page table은 이제 arbitrary size로 사용 가능하다고 하자.

→ memory 내부의 free space 찾는 것이 더 복잡할 수도 있음.

⇒ Btg page, segmentation 모두 단점 존재

→ 단점 극복 방법?



## ▼ Multi-Level Page tables → page table 나눠서 계층적으로 관리

### ▼ page table

page-sized unit에 딱 맞게 들어가도록 page table 조각냄

- 만약 PTE의 전체 page가 invalid할 경우 page table의 모든 page가 할당되지 않음

### ▼ page directory

page  
table  
관리

page table의 page가 어디에 있는지 or page table의 전체 page 중 no valid page를 포함하고 있는 지에 대해 알려줌

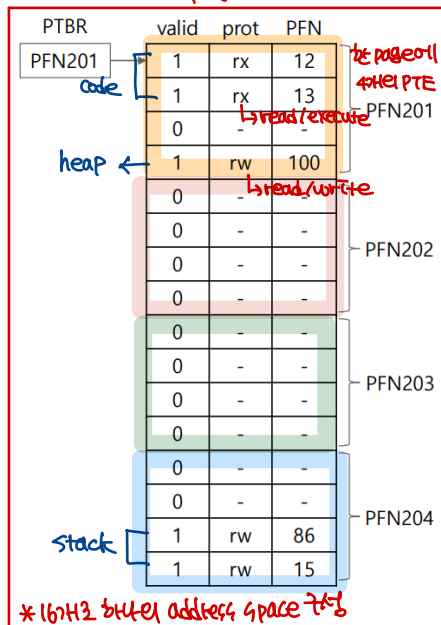
\*page 단에서 이렇게 정함.

Page Table				
PFN	valid	prot	present	dirty
10	1	r-x	1	0
-	0	r-x	0	-
-	0	-	-	-
-	0	-	-	-
23	1	rw-	1	1
-	0	-	-	-
-	0	-	-	-
-	0	-	-	-
-	0	-	-	-
-	0	-	-	-
-	0	-	-	-
-	0	-	-	-
-	0	-	-	-
-	0	-	-	-
28	1	rw-	1	1
4	1	rw-	1	1

### page table 내부에서 일정 공간을 heap 공간으로 지정하는 방식

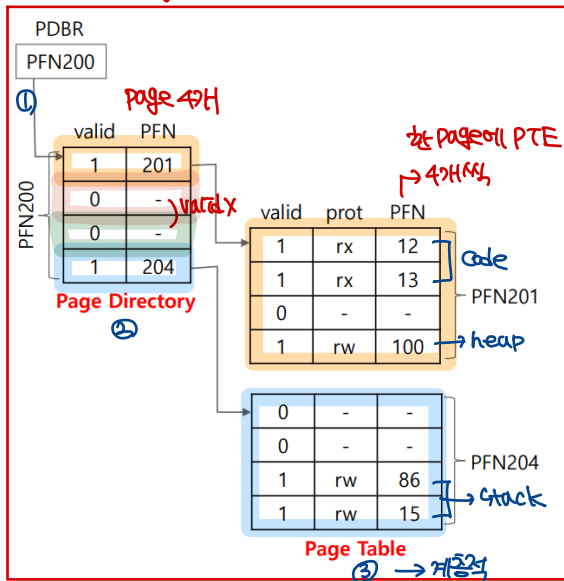
▼ Multi-level page tables ⇒ real 시스템에서는 이것보다 더 절약 가능

## linear page table



- \* 16H2 bit의 address space 찾기
- \* PAGE 4H 사용
- \* 8217에2 Linear

## Two-Level page table (multi level)



- \* 사용하지 않는 것의 entropy적 의미
- \* Page 기하
- \* 물리적으로 treat하지 않아도 됨.

• address translation  $\Rightarrow$  pseudo code

```

VPN = (VirtualAddress & VPN_MASK) >> SHIFT
(Success, TlbEntry) = TLB_Lookup(VPN)  $\rightarrow$  찾은 VPN에 대하여 TLB
if (Success == True) // TLB Hit
    if (CanAccess(TlbEntry.ProtectBits) == True)
        Offset = VirtualAddress & OFFSET_MASK
        PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
        Register = AccessMemory(PhysAddr)
    else
        RaiseException(PROTECTION_FAULT)

else // TLB Miss -> TLB linear와 다른 코드임
    //first, get page directory entry
    //Vpn -> page dir index, page table index 두 가지로 구성
    PDIndex = (VPN & PD_MASK) >> PD_SHIFT
    PDEAddr = PDBR + (PDIndex * sizeof(PDE))
    PDE = AccessMemory(PDEAddr)  $\Rightarrow$  page dir entry
    if (PDE.Valid == False)
        RaiseException(SEGMENTATION_FAULT)  $\rightarrow$  page fault
    else
        // PDE is valid: now fetch PTE from page table
        PTIndex = (VPN & PT_MASK) >> PT_SHIFT
        PTEAddr = (PDE.PFN << SHIFT) + (PTIndex * sizeof(PTE))
        PTE = AccessMemory(PTEAddr)  $\Rightarrow$  page table entry
        if (PTE.Valid == False)
            RaiseException(SEGMENTATION_FAULT)
        else if (CanAccess(PTE.ProtectBits) == False)
            RaiseException(PROTECTION_FAULT)
        else
             $\rightarrow$  address translation  $\Rightarrow$  TLB hit.
            TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
            RetryInstruction()

```

**TLB Hit**  
 $\rightarrow$  이전에 본 pseudo code와 동일함.

**page dir entry 찾기**

**TLB MISS**  
 $\rightarrow$  더 복잡해짐

**page table entry 찾기**

$\rightarrow$  memory 접근 2번

- 단점
- Time-space trade-off : 시간 Good  $\rightarrow$  공간 Bad, 공간 Good  $\rightarrow$  시간 Bad...
    - TLB miss  $\rightarrow$  memory로의 load 명령어 두 번이 있어야 함 (access memory 두 번)
      - multi page table로부터 정확한 translation 정보를 얻기 위함
      - 하나는 page dir, 하나는 PTE 자체  $\rightarrow$  메모리 access 두 번
  - Complexity (TLB miss일 때 처리해야 할 게 너무 많음)
    - page-table 검색을 처리하는 hw, OS 모두 linear page table보다는 더 많은 작업이 소요됨.
- $\Rightarrow$  OS 입장에서 복잡함

## ▼ example

- address space
  - 16KB address space with 64-byte pages
    - 14 bit addressing : 8bits → VPN, 6 bits → offset
  - 각 PTE : 4bytes (64/4) , 한 page 내에 (6개의 PTE 가능)
- page table

### 1. Linear page table

- $2^8$  (=256) tables
  - $2^8 \times 2^2 = 2^{10} \rightarrow$
- $256 \times 4\text{byte} = 1\text{KB}$

### 2. Multi-level page table

- 각 page → 16 PTE (or 6 PDE) ⇒ 4 bits 필요

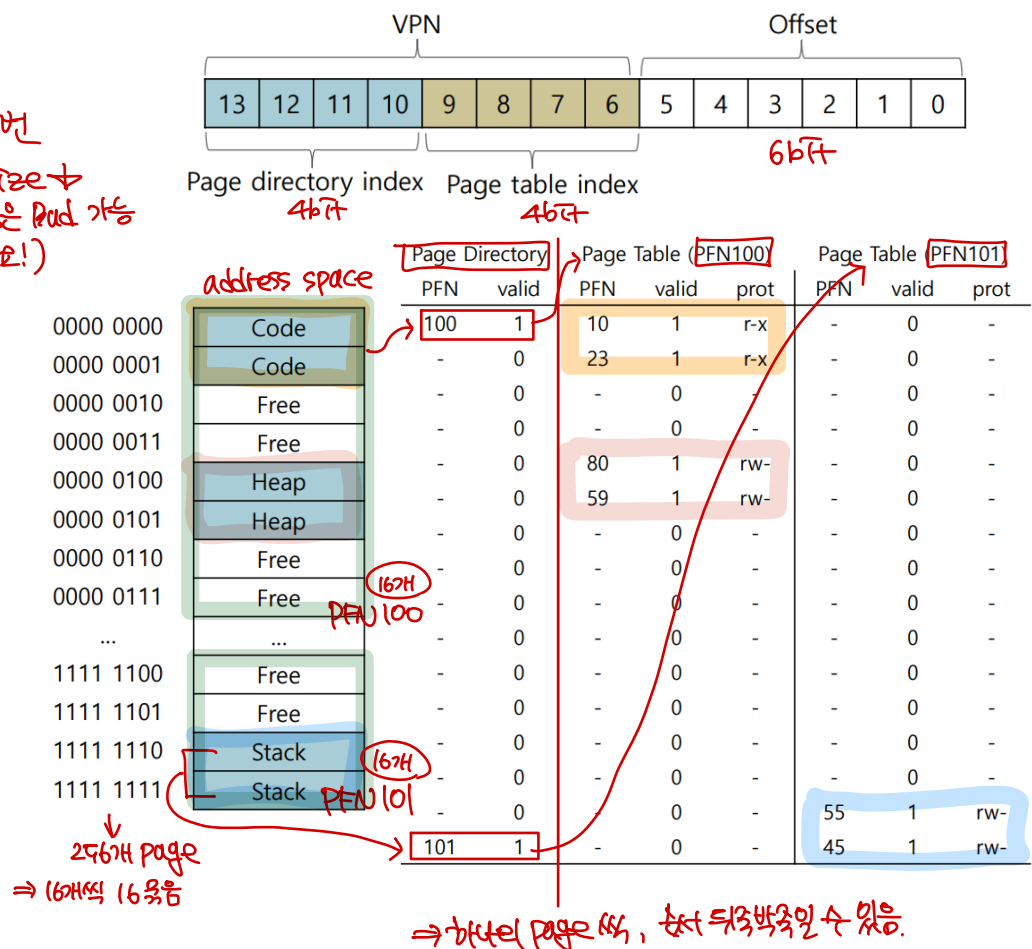
\* address translation

① PDI의 PFN 접근

② PTE의 PFN 접근

⇒ memory access 두 번

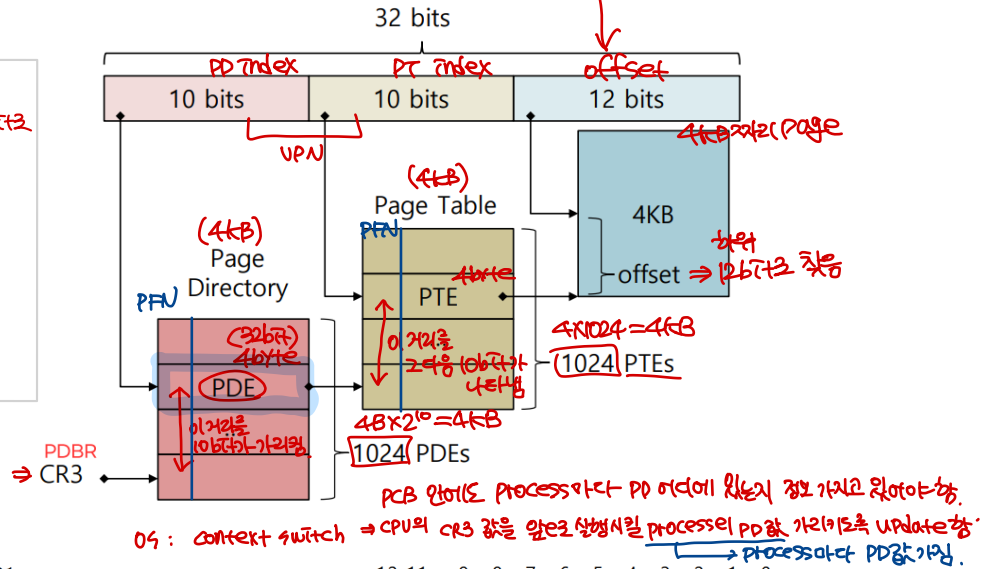
⇒ multilevel : page size ↓  
But, 성능 bad 가능  
(TLB hit가 중요!)



## ▼ x86-32 (2-level paging) ⇒ 실제로 구현된 내용 확인해볼자!

$$4\text{KB} = 2^2 \times 2^{10} = 2^{12}$$

- address translation
- ① CPU: CR3가 가리키고 있는 process의 PD에 대해서 10비트 indexing
  - ② address translation 하고자 하는 PTE가 해당 PDE 안에 다 있음.
  - ③ 그 다음 10비트 address translation 하고자 하는 PTE 찾기
  - ④ 찾은 PTE의 PFN이 address translation 하고자 하는 주소로 포함된 page 찾기.



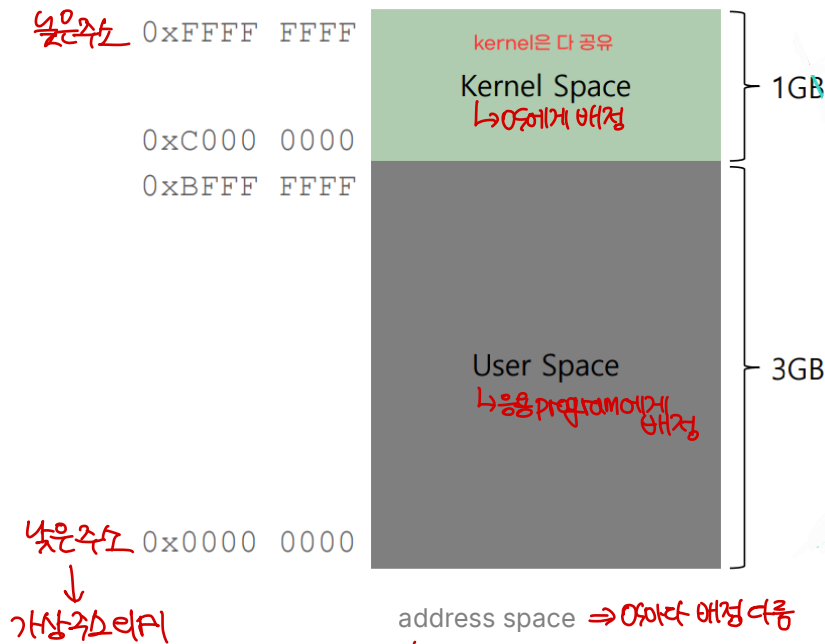
PDE

31	12	11	9	8	7	6	5	4	3	2	1	0
Page Table Address (PFN)												
				G	S	D	A	PCD	PWT	U/S	R/W	P

PTE

31	12	11	9	8	7	6	5	4	3	2	1	0
Physical Page Address (PFN)												
				G	PAT	D	A	PCD	PWT	U/S	R/W	P

PDE and PTE



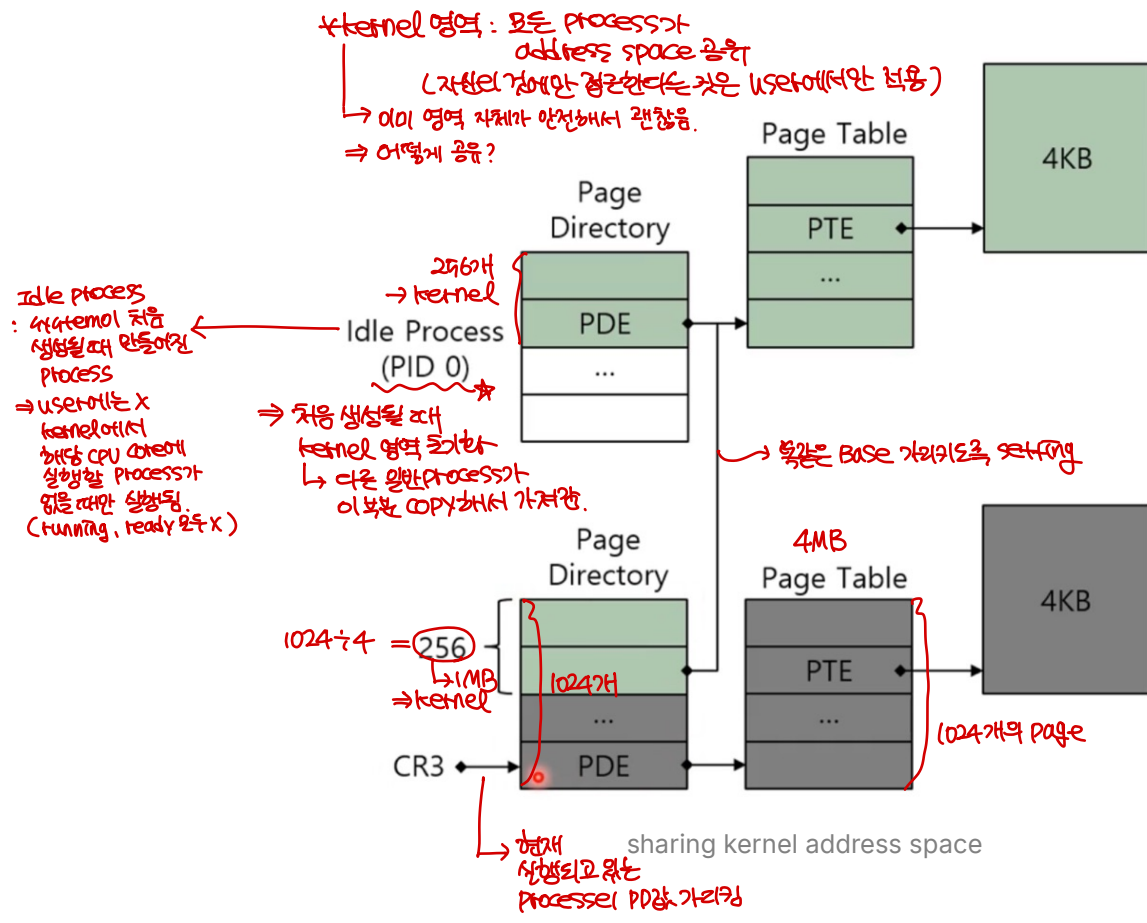
기본 page: 4KB

page entry 개수:  $2^{10}$ 개

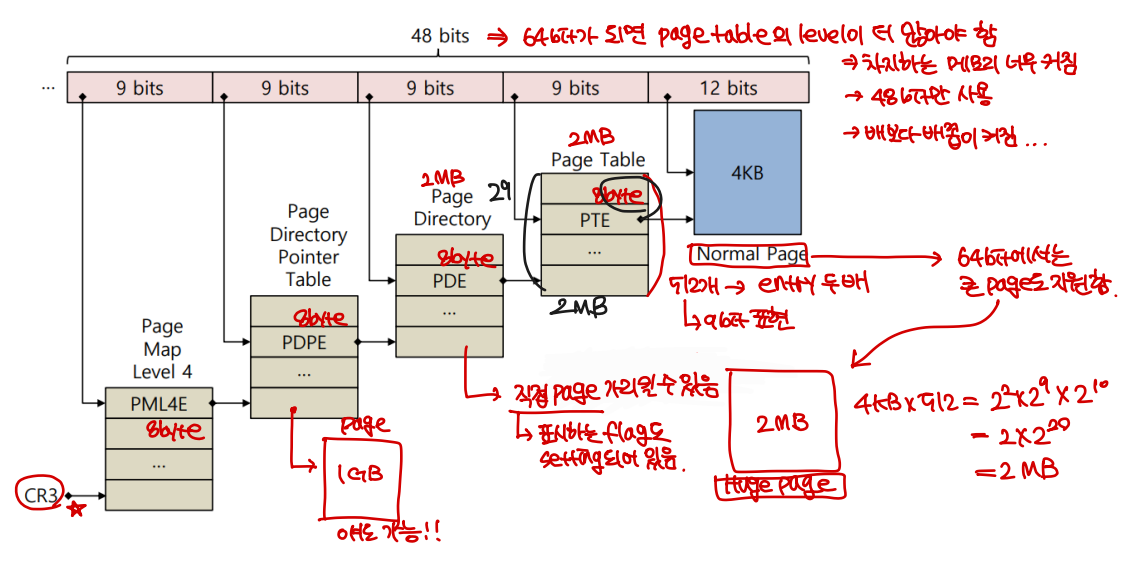
⇒ page 크기: 4MB

⇒ 100만 page table이 또  $2^{10}$ 개

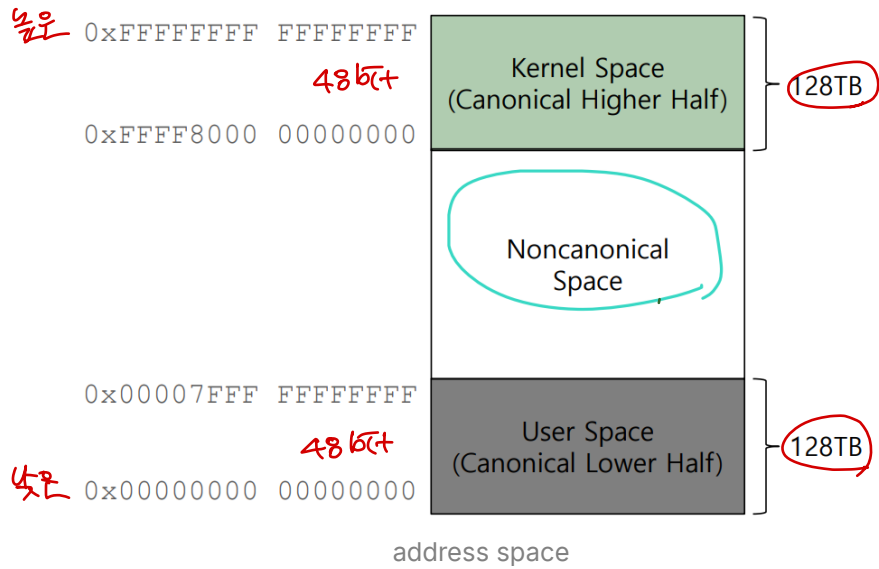
⇒  $4\text{MB} \times 2^{10} = 4\text{GB}$  ★



▼ x86-64 (4-level paging) 이번엔 64비트! ⇒ 512GB ★







\* Summary

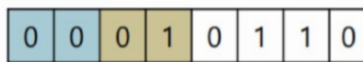
### Example

- 8-bit addressing



- 16-byte page (20 → 4672222)
- 4-byte PDE/PTE (6byte)

- Virtual address: 22



- Physical address: 214

