11. Multi-Level Page Tables

↳ page table 작게 관리하기 위해서 제안된 방법 (효율적으로!)

▼ Paging ⇒ page table 크기 계산해보자

▼ Linear page table (하나의 큰 덩어리로 갖자)

- page table : 너무 커서 메모리 소모량이 많음
  - abstraction을 적용하기 위한 memory 낭비도 큼 → cpu에 있기 어려움
- example
  - 32-bit address space with 4KB pages
  - 4byte PTE ★
  - 4MB per page table

offset : 12bit (2^{12}B)

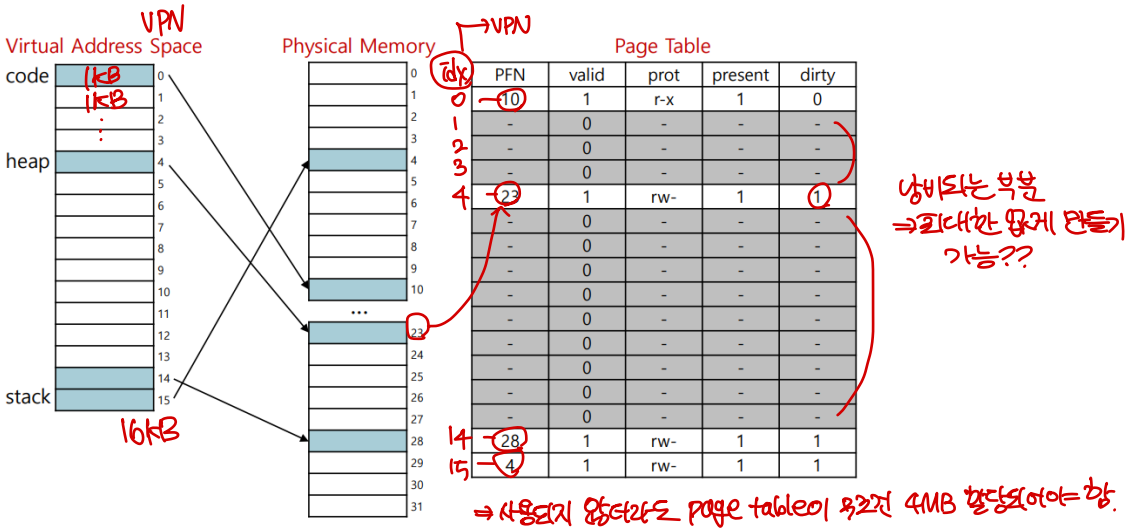
VPN : 20bit → page개수 2^{20}

page table 크기 : 2^{20} x 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 ⇒ 해결하기 위해서 page 사이즈만 키워보자!!
  - page 사이즈 커짐 → page table entry의 정보는 작아짐 → cache, heap
  - example
    - 32-bit address space with 16KB pages
    - 1MB per page table
  - Big page → 각 page 내에 낭비되는 놈 존재
    - internal fragmentation
      - process가 필요한 양보다 더 큰 메모리가 할당되어 공간 낭비 발생
      - process : 실제로 사용하지 않는 메모리 영역 가지고 있게 됨

↔ external fragmentaion : 메모리는 남는데 너무 쪼개져서 사용 불가한 경우

⇒ 대부분의 system : 보통 상대적으로 작은 page size 사용

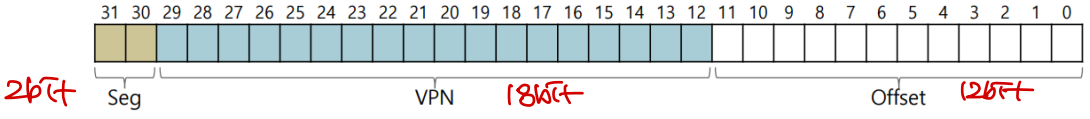
- 4KB in x86, 8KB in SPARCv9

② ▼ Hybrid approach ⇒ paging, segment 둘 다에 사용해보자!

- paging and segments
  - 각 segment 마다 하나의 page table
  - 3개의 base/bound pair → segment를 기본적으로 3개로 나눠짐다는 뜻
    - base register : segement마다 page table의 물리적인 주소를 가지고 있음
      - segment 스스로 point x
    - bounds register : segement 내부 최대 vaild page의 값을 가지고 있음

⇒ linear page table에 비해 memory saving 효과가 큼

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




```
SN = (VirtualAddress & SEG_MASK) >> SN_SHIFT
VPN = (VirtualAddress & VPN_MASK) >> VPN_SHIFT
AddressOfPTE = Base[SN] + (VPN * sizeof(PTE))
```

- 32-bit address space with 4KB pages
- 4 segments
  - 00 : Unused
  - 01 : Code
  - 10 : Heap
  - 11 : Stack

- 단점
  - ① 크지만 부족하게 사용되는 heap : 낭비되는 page가 많음
    - heap에 대해서는 linear한 page table로 관리해야 함
    - but, page table이 가변적으로 변해버림
    - free → waste 발생 / allocate → 더 큰 공간 필요

    ⇒ memory copy overhead가 발생할 수 있음
  - ② external fragmentation

    - page table은 이제 arbitary size로 사용 가능하다고 하자.

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

⇒ Big page, segmentation 모두 단점 존재

→ 단점 극복방법?

▼ Multi-Level Page tables

▼ page table

pase-sized unit에 꽉 맞게 들어가도록 page table 조각냄

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

▼ page directory

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

*page 단위에서 이렇게 관리.

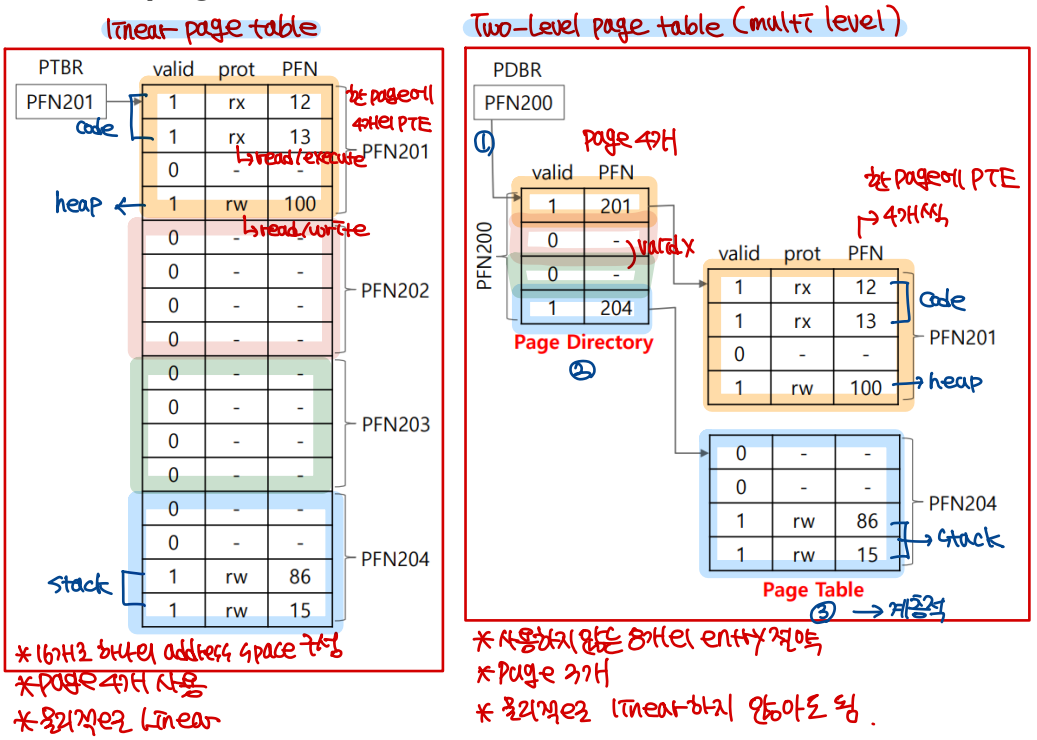
Page Table

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

0 1 2 3 4 5 6 code heap stack

page table 내부에서 일정 공간을 heap 공간으로 지정하는 방식

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



- address translation ⇒ pseudo code

```
VPN = (VirtualAddress & VPN_MASK) >> SHIFT
(Success, TlbEntry) = TLB_Lookup(VPN) → 찾은 VPN에 대해서 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) ⇒ page dir entry
  if (PDE.Valid == False)
    RaiseException(SEGMENTATION_FAULT) → 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)
    if (PTE.Valid == False)
      RaiseException(SEGMENTATION_FAULT)
    else if (CanAccess(PTE.ProtectBits) == False)
      RaiseException(PROTECTION_FAULT)
    else
      TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
      RetryInstruction()
```

TLB HIT → 이전에 본 pseudo code랑 동일함.

TLB MISS → 더 복잡해짐

page dir entry 찾기

page table entry 찾기

→ memory 접근 2번

→ address translation ⇒ TLB hit.

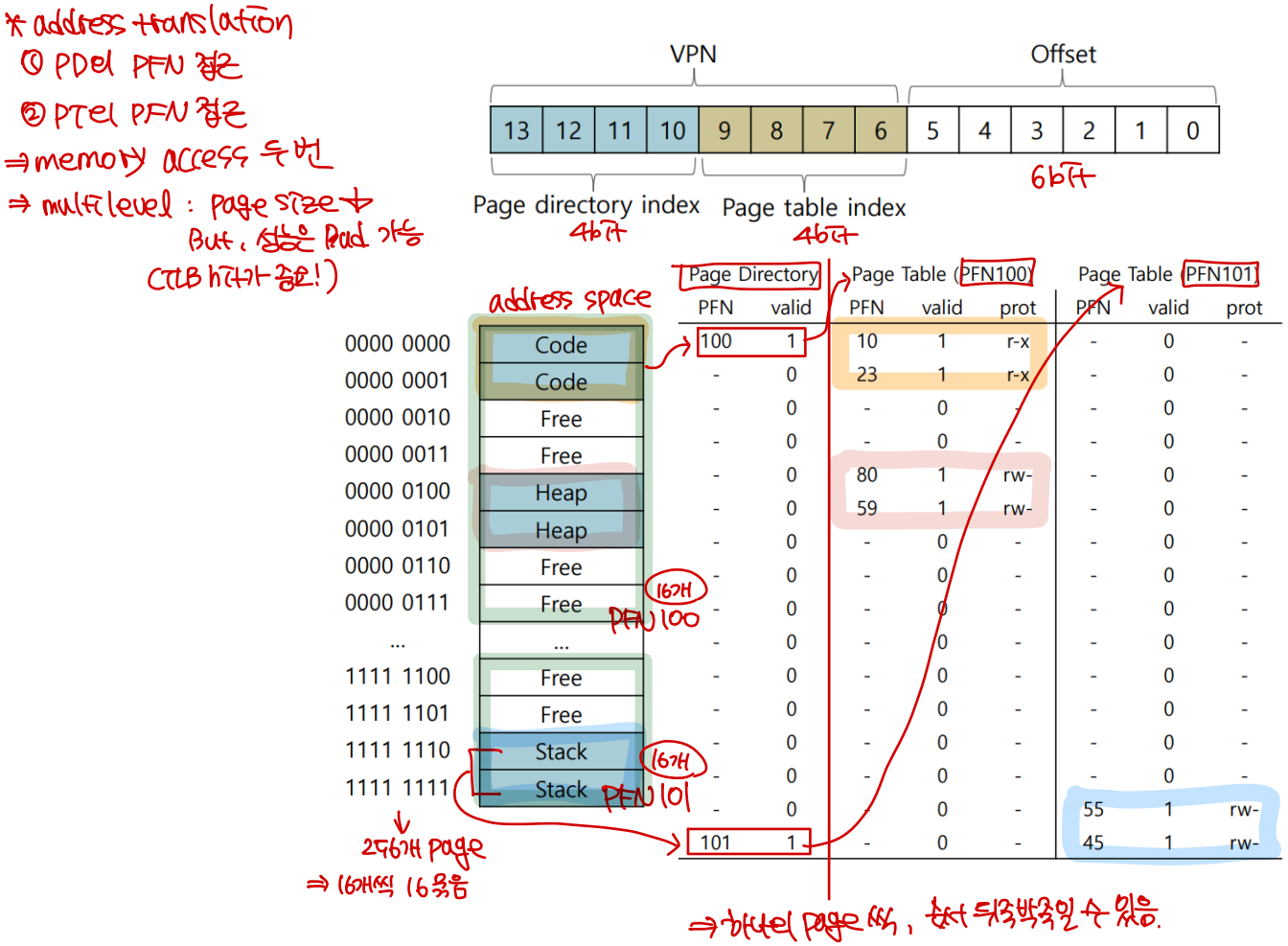
단점

- Time-space trade-off : 시간 Good → 공간 Bad, 공간 Good → 시간 Bad...
  - TLB miss → memory로의 load 명령어 두 번이 있어야 함 (accessmemory 두번)
    - multi page table로부터 정확한 translation 정보를 얻기 위함
    - 하나는 page dir, 하나는 PTE 자체 → 메모리 access 두 번
- Complexity (TLB miss일 때 처리해야될게 너무 많음)
  - page-table 검색을 처리하는 hw, OS 모두 linear page table보다는 더 많은 작업이 소요됨.

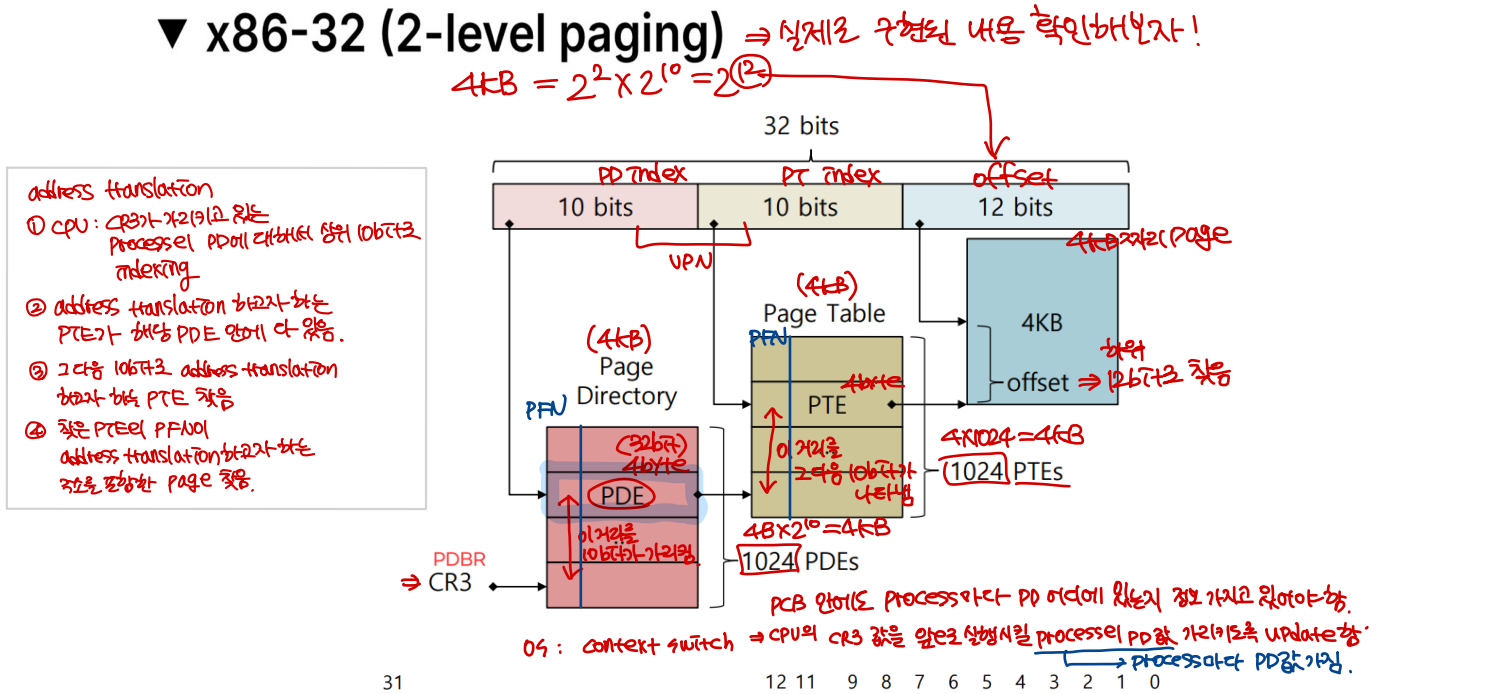
⇒ OS 입장에서 복잡함

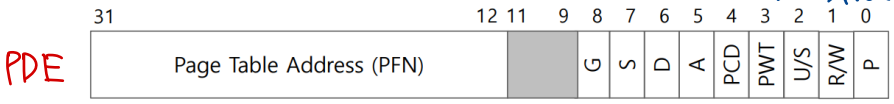
▼ example

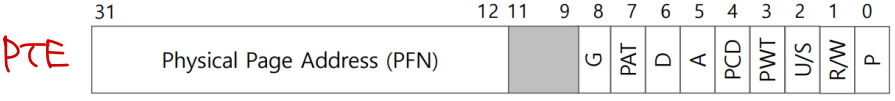
- address space
  - 16KB address space with 64-byte pages
    - 14 bit addressing : 8bits → VPN, 6 bits → offset
  - 각 PTE : 4bytes (64/4)
- page table
  1. Linear page table
     - 2^8 (=256) tables
     - 256 X 4byte = 1KB
  2. Multi-level page table
     - 각 page → 16 PTE (or 16 PDE)



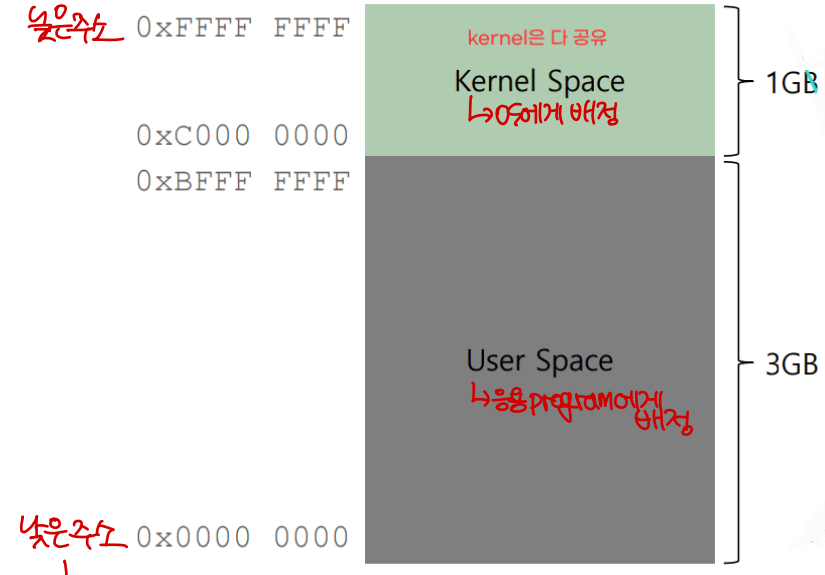
▼ x86-32 (2-level paging)





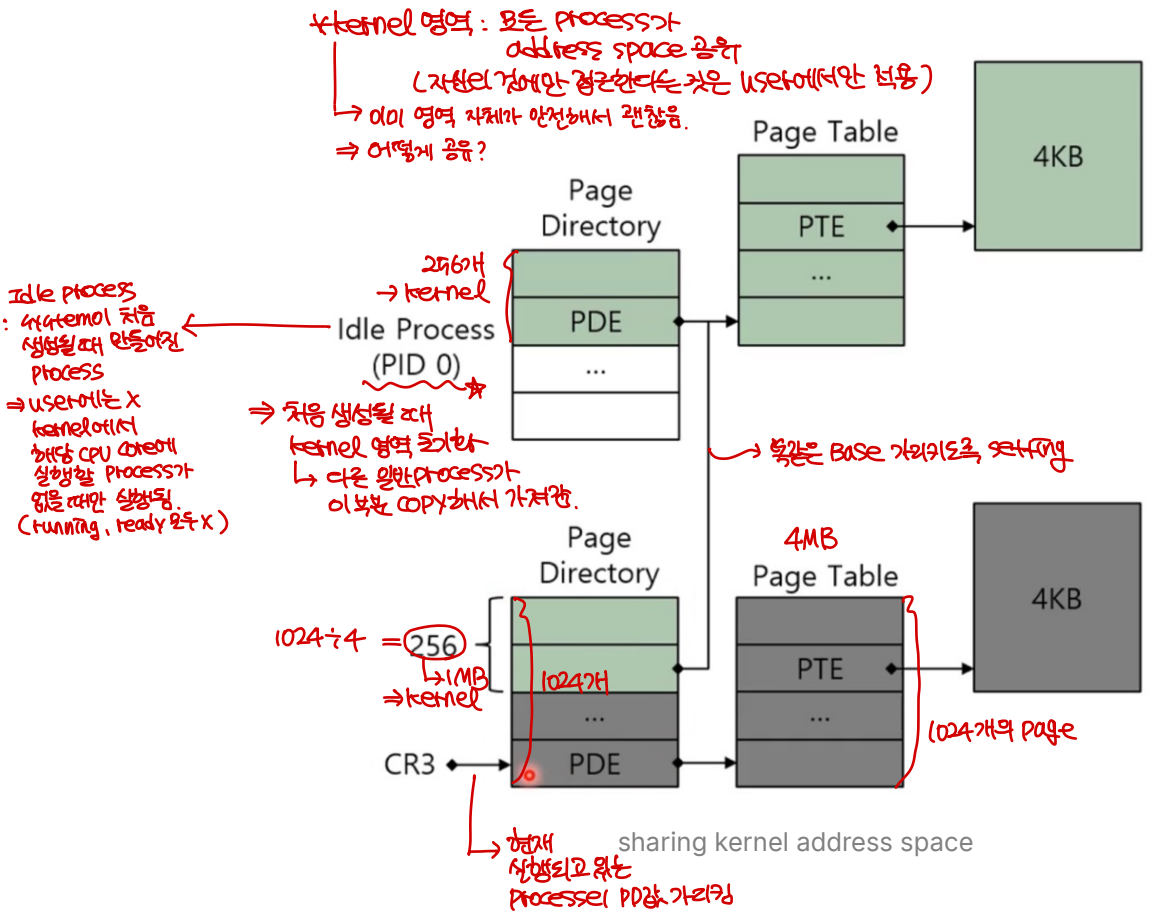


PDE and PTE

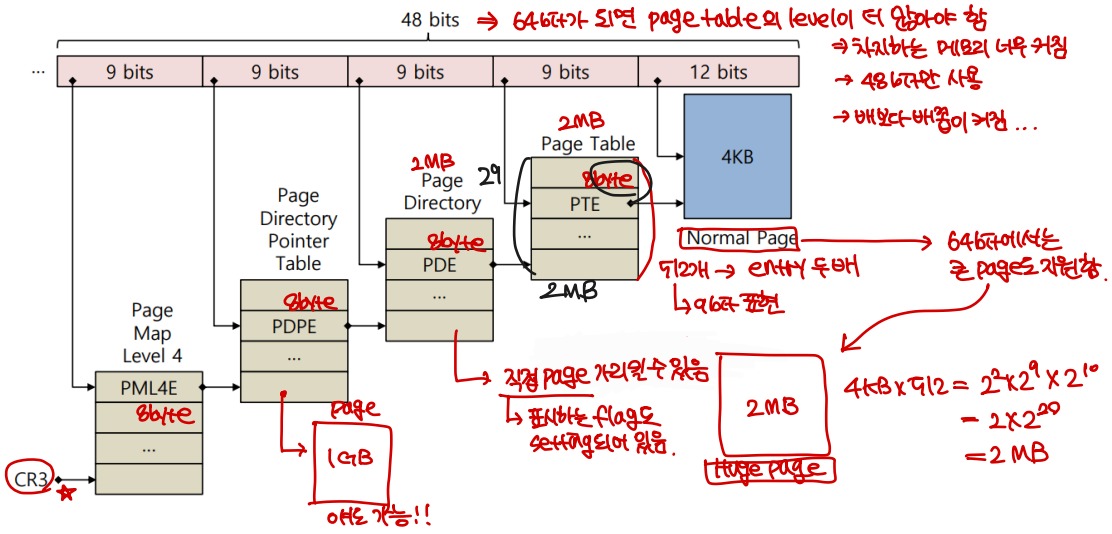


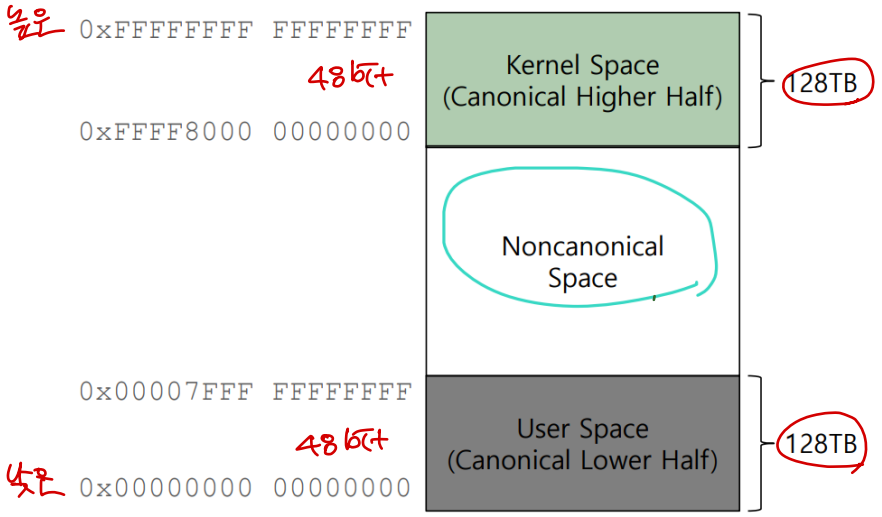
address space ⇒ 주소 해석 단위

11. Multi-Level Page Tables



▼ x86-64 (4-level paging)





address space