paging : 고정된 크기의 slot (page frame)의 배열로 physical memory 나눔.
↳ page table : VPN, PFN mapping하는 table
↳ 66가 address → 64 byte 표현 (2^6)
16 byte page frame (offset 4개)

⇒ page 얼마나 큼?
page table 어디에 저장?

09. Paging

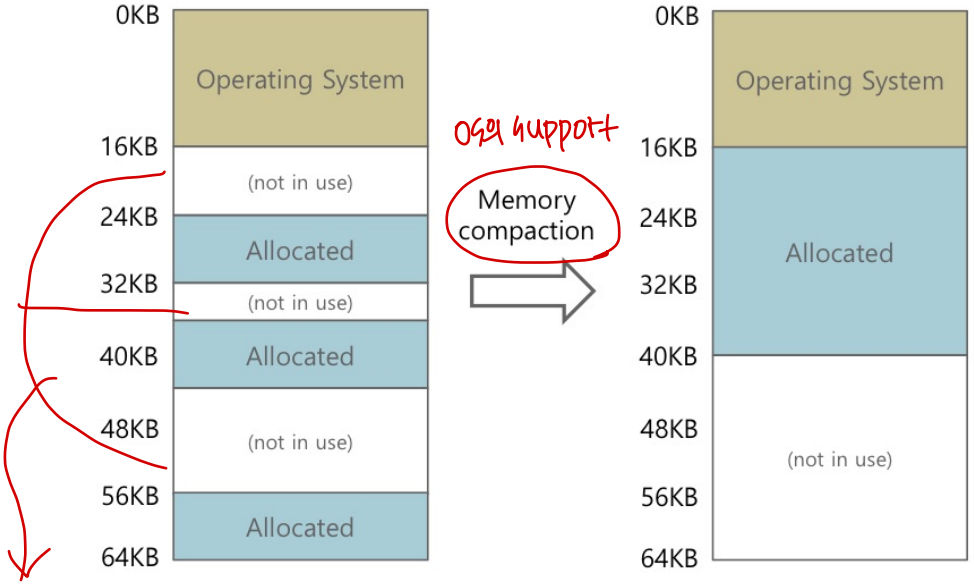
▼ Space Management

▼ Segmentation

memory compaction

크기가 가변적인 메모리 조각 ⇒ overhead 너무 큼 → 관리가 어려움

- 가용 가능한 공간이 조각 나있음 → 메모리가 큰 친구는 사용 못 함



- external framentation ☆
  → 고정된 크기를 가지고 있어야 해결
  → 다른 방법은 없을까? ☆

⇒ ***Paging!!***

(작아도 고정 하나, 크면 page frame 2개)
↦ memory 할당할 때 무조건 page frame 단위로 할당

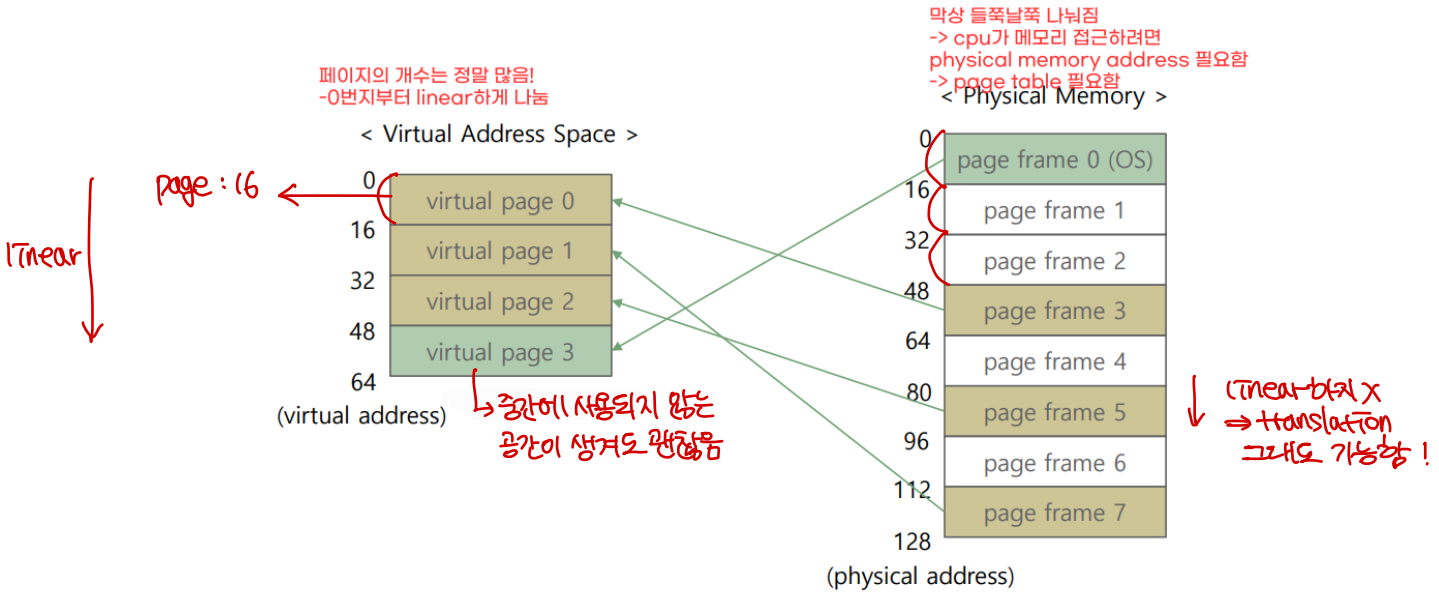
▼ Paging

page frame(꽤 작게 고정된 size의 slot 배열)로 physical memory 나눠주고 보여줌

(page frame에서 말하는 똑같은 크기로 나누는 것은 segment와 다른 나누기임)

장점
- address space의 효율적인 abstraction 지원 가능
  - process가 address space를 어떻게 사용하는 지에 상관 없이 가능
    ↳ heap, stack에 대한 정보 필요 X
- free-space 관리가 쉬워짐

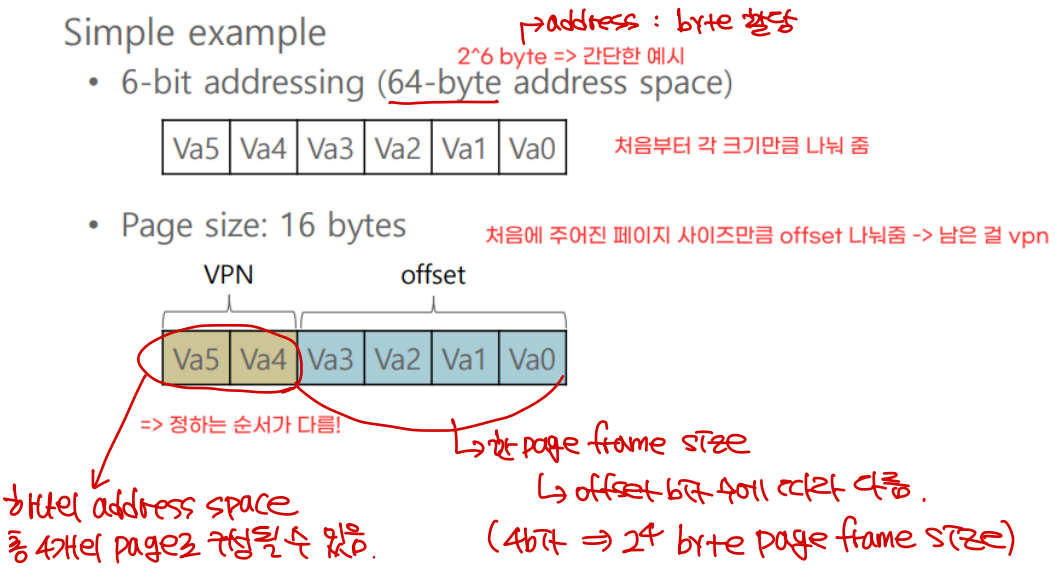
- example

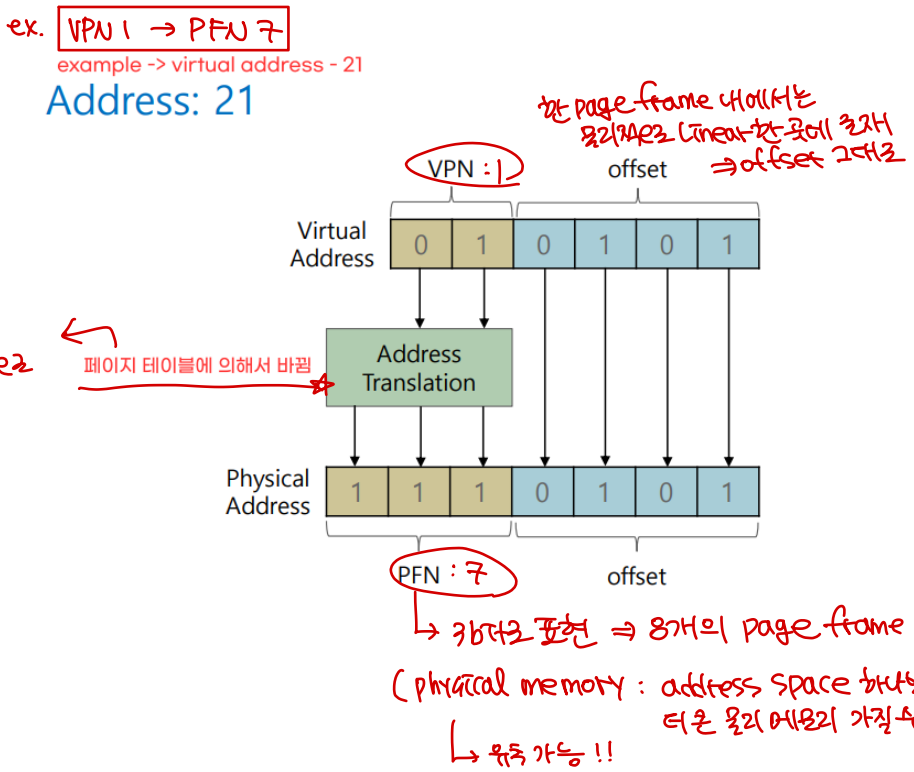


segement로 나누어진 physical memory를 일정한 크기로 나눔 → virtual page에 할당

▼ Address Translation

- Virtual address
  - Virtual page number + offset (CPU마다 할당된 bit 수가 다를 수 있음) ⇒ 몇 bit씩 할당할지는 CPU가 정함.



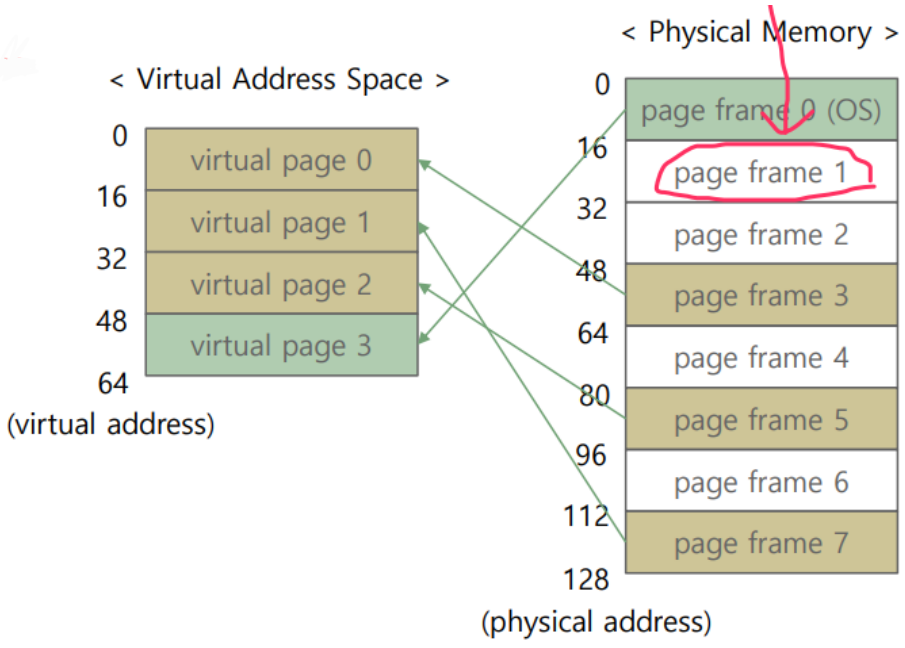


▼ Page table

▼ page table

- address space의 각 virtual page에 대한 address translation을 저장하기 위한 process 단위 자료구조
- VPN → PFN index라고 생각하기 → 따로 table이 마련되어야 하는 것은 x

page table
VP0 → PF3
VP1 → PF7
VP2 → PF5
VP3 → PF0



- Linear page table
  - 빈 공간이 낭비될 수 있는 table임 → 그렇게 좋은 방법은 아님
  - 더 좋은 자료구조를 다음 chapter에서 볼 예정

▼ page table에 대한 question

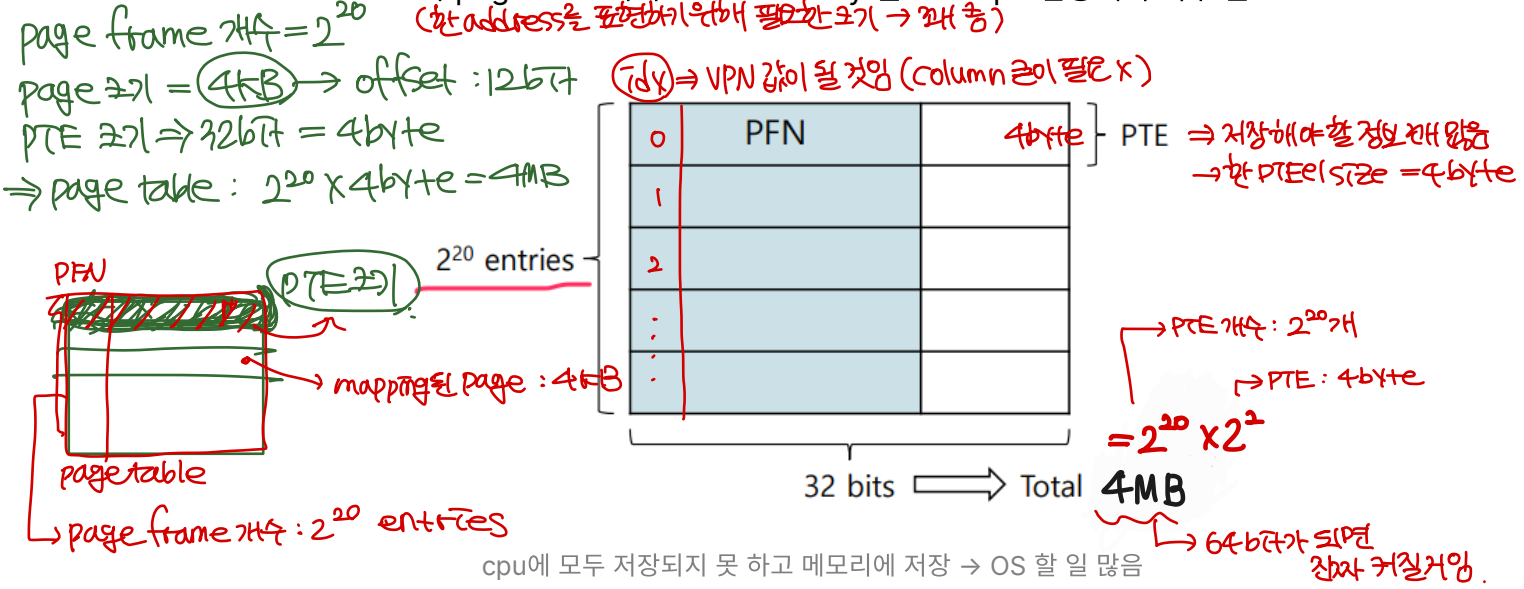
1. page table은 얼마나 커지지?

- page table : 너무 커질 수도 있음

  ex) 20bit VPN + 12bit offset = 32bit

  20bit VPN = 각 process마다 2^20 translation

  - 각 PTE(page table entry)마다 4byte(32bit)
  - 각 page table마다 4MB memory 필요 ⇒ cpu 입장에서 너무 큼



cpu에 모두 저장되지 못 하고 메모리에 저장 → OS 할 일 많음

2. page table은 어디에 저장되지?

- OS가 관리하는 physical memory에 각 process마다 page table 저장됨
  - hw에게 너무나도 큰 사이즈

3. page table의 다른 요소는 무엇이 필요하지?

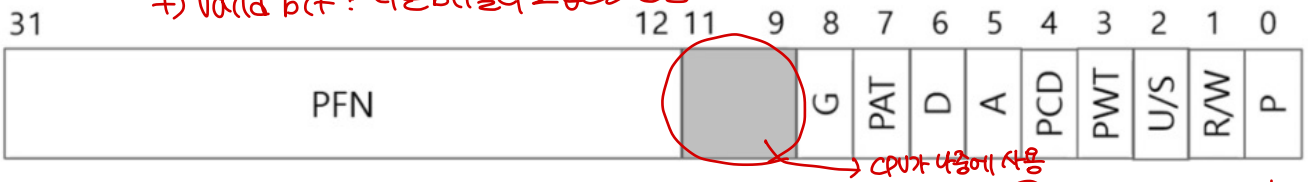
- PTE(page table entry)
  1. **Vaild bit** : address translation 가능 여부를 0,1로 표현
     - 모든 unused space → invailed로 표시되어 있음
     - sparse address space 지원 가능
       - sparse address space : 공백을 포함하는 virtual address space
  2. **Protection bits** : read, write, execute 등 여러 권한을 표현
  3. **Present bits** : vaild but mapping 가능 여부 0,1로 표현

- page frame, physical memory는 꼭 필요한 것만 미리 mapping
  a. 너무 오래 걸림
  b. 낭비 심함
  ⇒ 접근 시에 필요할 때만 mapping 하는 구조 ★
- page가 disk 내부의 physical memory에 할당되어 있는지

4. **Dirty bit** : page frame에 대해 지금까지의 사용 여부 저장하는 bit
   - OS : page frame이 부족할 경우 당장 필요한 놈한테 할당함
     a. 가급적 한 번도 page를 쓰지 않은 애한테서 할당 취소
     b. 만약 한 번이라도 쓴 애가 있다면 disk에 해당 process 저장
     → 일이 복잡함..
     → 이러한 여부를 저장하는 bit
5. **Reference bit** : 읽은 시점에 대해 판단
   - used by page replacement policy

example
- x86-32 PTE (Intel 예시 ver)
  - P: present bit → memory mapping 여부
  - R/W: read/write bit
  - U/S: user/supervisor bit → user mode or supervisor mode access 가능 여부
  - A: accessed bit → reference bit과 같음
  - D: dirty bit → mapping된 뒤에 operation 유무
  - PWT, PCD, PAT, G: hardware caching policy
  - PFN: page frame number

+) valid bit : 다른 bit들이 고장으로 판단



해당 사항은 cpu가 결정 → 메모리에 존재하는 정보를 관리하는 건 os ★ ⇒ address translation 가능!

4. paging이 system을 느리게 만들지는 않을까?

- hw(cpu) : 현재 running process의 page table이 어디에 있는지 반드시 알아야 함 ⇒ CPU 안에 있는 register가 memory 내부의 page table 가리킴
  → CPU가 정해놓은 format대로 OS가 table 모두 채워줌
  ⇒ address translation 가능
- 메모리 접근이 필요함

PTBR

- page-table base register(PTBR)
  - page table에 대한 시작 위치 physical address 가지고 있음.

  ⇒ address translation 느리지 않게 가능!
- cpu chip에 남지 않고 memory에만 남음 → 속도가 너무 높아져서 overhead 발생 가능

- address translation → CPU가 수행 (memory 접근 ⇒ 느림)

```
VPN = (VirtualAddress & VPN_MASK) >> SHIFT
PTEAddr = PTBR + (VPN * sizeof(PTE))
```

```
PTE = AccessMemory(PTEAddr)
if (PTE.Valid == False)
  RaiseException(SEGMENTATION_FAULT)
else if (CanAccess(PTE.ProtectBits) == False)
  RaiseException(PROTECTION_FAULT)
else
  offset = VirtualAddress & OFFSET_MASK
  PhysAddr = (PTE.PFN << SHIFT) | offset
  Register = AccessMemory(PhysAddr)

//VPN_MASK : 0x30 (110000)
//SHIFT: 4
```