12. Swapping

▼ Demand paging

swapping을 정의하기 전에 demand paging에 대해 명확하게 정의해야 함

→ demand paging : 필요한 부분의 page 영역만 물리적 메모리에 할당하는 방법 → 요청이 있을 때만!

⇒ OS : demand paging 사용

- mapping이 vaild하지 않은 entry 꽤 있었음 → 모든 address space 다 쓰지 않는다
  - OS : page에 접근할 때 page를 memory로 가져옴 → page mapping
    - memory에 접근하는 것에만 mapping(vaild) → multi level 장점 살릴 수 있음
  - 다 mapping하지 않기 때문에 page fault 발생
    - page fault : page가 invalid(not present)
      - vaild bit → 1이면 존재(vaild), 0이면 존재 x(invaild)
    - 실행 초반에 많이 발생 가능 → 해결하기 위해 prefetching 사용하기도 함
- prefetching
  - 접근할 page를 미리 mapping하여 짧은 시간 내에 메모리 잘 접근한다는 가정일 때의 예시
    - ex) page *P*를 위한 physical memory mapping 시에 page *P+1*도 미리 mapping(곧 접근할 놈임)

VPN VPN

▼ Swapping

▼ Physical memory

지금까지 physical memory 자원이 항상 풍부하다는 가정으로 얘기함

- 현재 실행 중인 많은 process에게 큰 address space 지원함
  - 모든 page가 physical memory에 있다고 가정
  - 큰 address space 지원하기 위해 OS는 memory hierarchy의 추가적인 level을 필요로 함
    - 지금 당장 실행에 필요한 놈들만 적재하고 안 쓰는 놈들은 보관

- 적재할 저장장치가 필요함 (ex. hard disk, SSD)

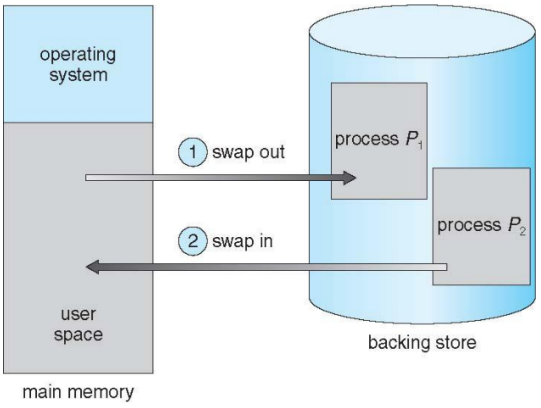
→ 실제로는 한정적, but address space는 꽤 크다

→ physical memory 부족함

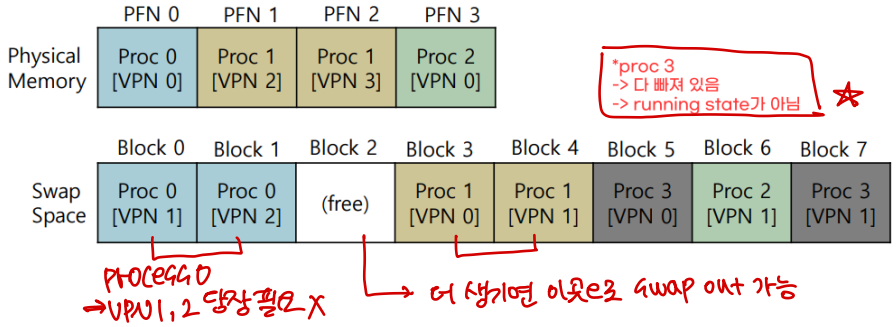
⇒ OS : 어떻게 더 크고 느린 장치를 사용하여 큰 virtual address space에 대해 환상을 제공??

▼ Swap Space

page 이동하기 위해 disk에 추가로 할당된 저장 공간 → 지금 당장 필요하지 않은 page 미리 빼놨다가 필요할 때 다시 또 실행



- **swap-out** : memory에서 빼내서 swap space에 저장
- **swap-in** : swap space에서 memory로 다시 저장
  - OS : page 크기만큼 swap space로부터 read/write 가능
  - OS : 주어진 page에 대한 disk address 기억할 필요가 있음



▼ Present Bit

swap in, swap out → 어떤 상태인지 제공해주는 bit

⇒ 해당 page가 memory 상에 있는지, 아닌지 알려줌

- TLB miss
  - h/w가 PTE 찾아볼 때, page가 physical에 존재하는지 찾아볼 수 있음
  - Present bit로 가능
    - 1: page가 physical memory에 존재
    - 0: memory에 존재하지 않고 disk 어딘가에 존재 → 즉, swap space에 존재!!

▼ Page fault

physical memory에 존재하지 않는 page를 접근할 때 발생

→ page의 present bit가 0으로 세팅되어 있을 때 발생

- **page fault handler**
  - swap out인 page 찾아서 swap in 해줌
  - OS가 어떻게 원하는 page 위치 찾음? (swap out되어 있는 놈이 swap space 어디에 있지?)

    → PTE 내부의 PFN으로 disk address 찾음(swap 공간의 offset을 의미하게 됨)

    → 기본 단위: 4KB (page 단위)

    → page fault가 발생할 때마다 전체 system이 상당히 멈춤.
  - disk I/O 작업이 완료되었을 때, OS는 present bit와 PFN update
    - 너무 큰 memory가 필요한 process의 경우 disk I/O 많을 수 있음
  - I/O 진행 중인 process → blocked state

- **HW**의 page fault control ⇒ Linear page table 가정! swap out 되었을 때의 경우!

```
VPN = (VirtualAddress & VPN_MASK) >> SHIFT
(Success, TlbEntry) = TLB_Lookup(VPN)
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
  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 if (PTE.Present == True) → present O
      // assuming hardware-managed TLB
      TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
      RetryInstruction()
    else if (PTE.Present == False) → present X
      RaiseException(PAGE_FAULT) ⇒ page fault
```

TLB 미스

TLB 미스

TLB 넣고 다시 시도

- **SW**의 page fault control ⇒ OS의 page fault handler

```
PFN = FindFreePhysicalPage()
if (PFN == -1) // no free page found
  PFN = EvictPage() // run replacement algorithm
DiskRead(PTE.DiskAddr, PFN) // sleep (waiting for I/O)
PTE.present = True // update page table with present
PTE.PFN = PFN // bit and translation (PFN) ★
RetryInstruction() // retry instruction
```

→ memory 어디에 적재? free한 공간 find

PFN 못 찾음

PFN 잘 찾음

→ Blocked state로 이동!!

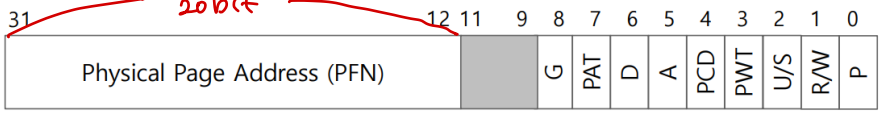
→ memory COPY → disk에서 swap space에서 가져옴.

→ PTE가 present
⇒ 다시 translation
→ TLB 찾음
⇒ TLB Hit

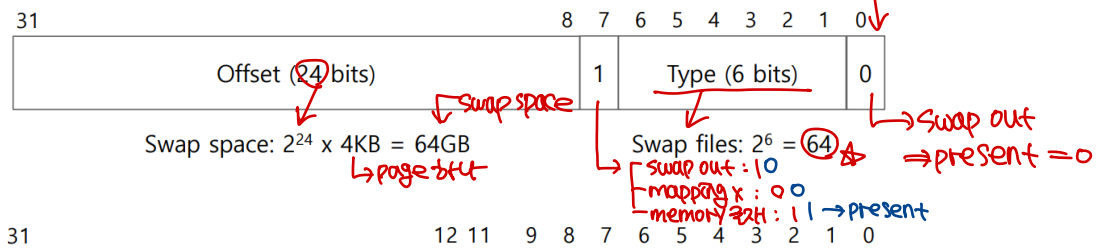
→ retry 2번

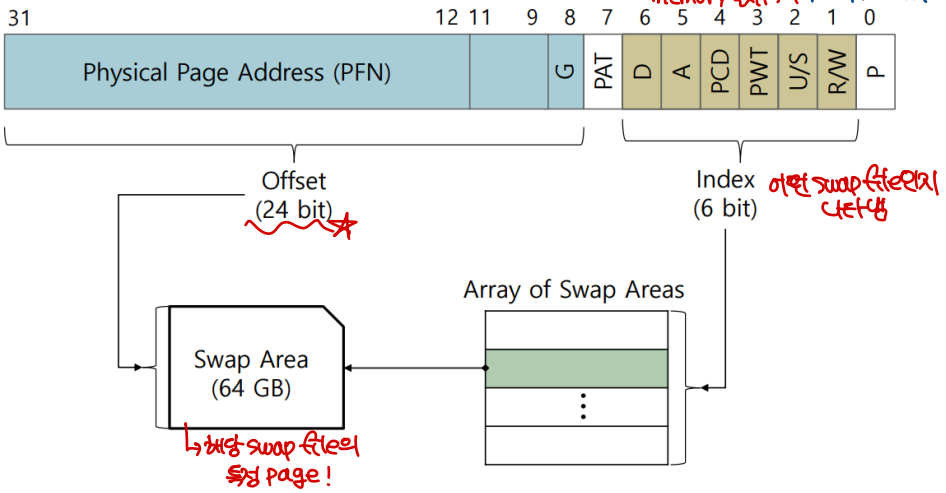
▼ Swap entry in Linux/x86-32

- PTE



- Swap entry





▼ Page Replacement

- memory가 full일 때
  - 하나 이상의 page를 swap out → OS가 넣으려는 새 page에 대한 공간 확보
- replacement는 실제로 언제 일어남?
  - HW(high watermark), LW(low watermark)
  - swap daemon (swap background process) :
    1. 사용 가능한 page가 적을 경우 → memory free
    2. 사용 가능한 page가 HW만큼 있을 때까지 page swap out
- 어떤 page를 eviction할지 어떻게 결정? → page replacement policy
  - 제대로된 알고리즘이 아니게 되면 disk의 speed로 program을 돌리는 현상이 발생할 수 있음

⇒ eviction 알고리즘 ⇒ 성능에 중요함!

▼ Page-Replacement Policy

1. Optimal replacement policy(비현실적인 ver)
   - 앞으로 가장 제일 사용되지 않을 page를 내보냄
     - 사실 응용 프로그램이 어떤 거에 접근할지 모르지만 안다고 가정해보자

Access (VPN)	Present?	Evict	Mapping
0	N		0
1	N		0, 1
2	N		0, 1, 2
0	Y		0, 1, 2
1	Y		0, 1, 2
3	N	2	0, 1, 3
0	Y		0, 1, 3
3	Y		0, 1, 3
1	Y		0, 1, 3
2	N	3	0, 1, 2
1	Y		0, 1, 2

2. FIFO
   - 처음 들어온 page를 가장 먼저 내보냄

Access (VPN)	Present?	Evict	Mapping
0	N		0
1	N		0, 1
2	N		0, 1, 2
0	Y		0, 1, 2
1	Y		0, 1, 2
3	N	0	1, 2, 3
0	N	1	2, 3, 0
3	Y		2, 3, 0
1	N	2	3, 0, 1
2	N	3	0, 1, 2
1	Y		0, 1, 2

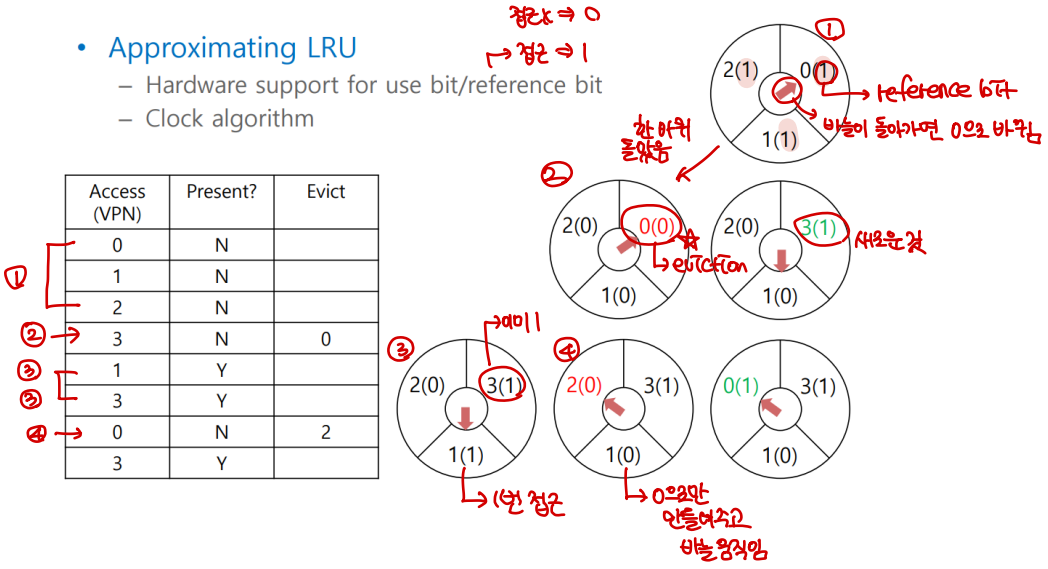
   - Belady's anomaly(변칙) : queue의 크기가 늘어났는데 hit rate 나빠짐
     - ⇒mapping 될 수 있는 page 개수 늘어났지만 page fault 더 많이 발생

3. Least-Frequently-Used(LFU) and Least-Recently-Used(LRU)
   a. LFU : 사용 빈도가 가장 낮은 애를 내보냄
   b. LRU : 최근에 젤 사용하지 않은 애를 내보냄

   ⇒ history 사용(frequency, recency...등)

Access (VPN)	Present?	Evict	Mapping
0	N		0
1	N		0, 1
2	N		0, 1, 2
0	Y		1, 2, 0
1	Y		2, 0, 1
3	N	2	0, 1, 3
0	Y		1, 3, 0
3	Y		1, 0, 3
1	Y		0, 3, 1
2	N	0	3, 1, 2
1	Y		3, 2, 1

- historical algorithm 구현해보자
  - 사용 빈도, 최근 사용을 찾기 위해 memory reference에 대한 약간의 연산을 좀 해야 함 ⇒ time stamp와 같이 추가적으로 필요한 것들이 존재
  - h/w support for time field
    - 이러한 정보들이 PTE마다 필요
    - system의 page 수가 많아질수록 많은 time field 스캔하는 것이 비쌈
- Apporximating LRU(LRU 근사 알고리즘)
  - h/w support → PTE의 reference bit 사용 ⇒ 1 bit로 구현 가능
  - clock algorithm 사용



- dirty page 고려해보자
  - page가 수정된 적이 없을 때 추가적인 I/O 없이 다른 목적으로 재사용하기 간단할 수 있음
    - swap 공간에 저장할 대 swap in은 read, swap out은 write로 악용 가능

- code segment로 이용된 곳은 read only임
- 이런 page에 대해서는 eviction 할 때 write 작업 없이 할당 가능

⇒ 가공적 I/O 작업 덜 해서 성능 높이려는 시도를 한다!!