4. Operation System (Virtual Memory)

https://myvelop.tistory.com/202?category=946865

https://eunajung01.tistory.com/87

---

1. 가상 메모리

< 탄생 배경 >

⇒ 가상 메모리를 사용하기 전에는 프로세스가 실행되려면 코드 전체를 물리적 메모리에 로드함

⇒ 메모리 용량보다 더 큰 프로그램은 실행이 힘들었음. (물론 overlays기법을 사용할 순 있다.)

⇒ 하지만 실제로는 코드량이 아무리 많아도 코드의 일부에서 대부분의 시간을 사용함

⇒ 또, 작은 양의 주소 공간을 사용하는 경우가 많음

⇒ 코드 전체를 물리적 메모리에 load하는 방식은 매우 비효율적이다.

※ 정의

⇒ 운영체제가 온전히 관리하는 영역, 물리 메모리에서 논리적 메모리의 분리!

⇒ CPU가 실제로 접근 및 필요로 하는 부분, 즉 실행에 필요한 일부만 메모리에 올림

⇒ 나머지는 비용 적지만 용량은 큰 디스크 Disk, 스왑 영역 Swap Area에 놓는다.

- page 단위로 swap in & swap out

⇒ 이를 통해 프로세스 전체가 물리적 메모리에 있는 것처럼 수행된다.

⇒ 물리적 메모리인 RAM의 자원이 훨씬 많이 있는 것처럼 보이며, 물리적 메모리에 소규모의 정보만 올려도 충분히 프로세스를 수행 가능하다. (더 많은 프로그램을 동시에 실행 가능)

- OS는 CPU가 필요한 자원을 받아 알아서 Swap 하며, 마치 전체 프로세스가 있는 것처럼 수행!

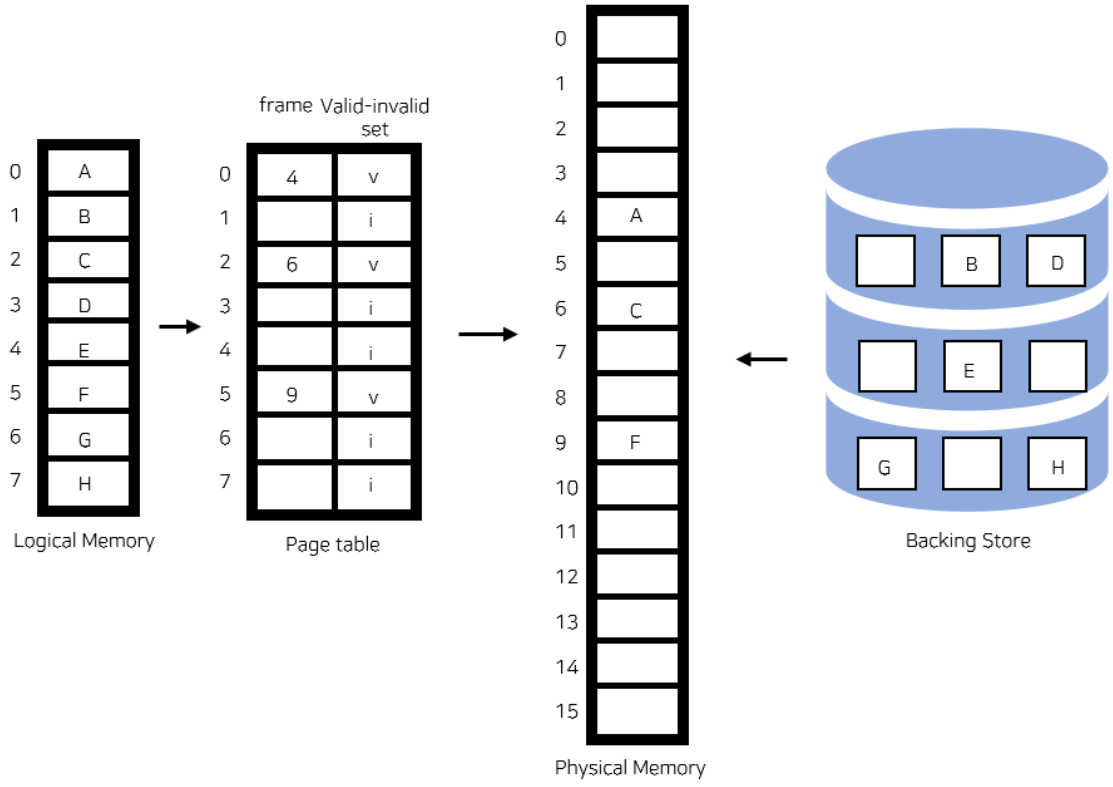
⇒ 또한, 효율적인 프로세스 생성도 가능하다. (바뀐 부분이 없다면 복사하지 않고 그대로 이용..)

⇒ 이를 통해, 비교적 값이 비싼 물리적 메모리를 더 효율적으로 사용!

⇒ 단, CPU가 memory에 올라와 있지 않은 영역에 대해서 접근할 때 어떻게 할지를 처리해야 함..

---

2. Demand Paging



(v = 이미 메모리에 있음을 나타내는 valid, i = 메모리에 없는 invalid)

⇒ 실제로 필요한 page만 메모리에 올리는 기법, 접근하는 page만 I/O 발생!

⇒ I/O와 Memory 사용량을 줄일 수 있고, 빠른 응답 시간과 더 많은 사용자의 수용을 보장!

※ Lazy swapper

⇒ 특정 page가 필요해질 때 까지는 메모리로 swap하지 않도록 하는 모듈

※ valid-invalid bit

⇒ Demand paging : page table에서 해당 page가 메모리에 있는지를 나타내는 valid-invalid bit 사용

⇒ bit가 **invalid**라면 사용되지 않는 주소 영역이거나 페이지가 물리적 메모리에 없음

- Process 내 어떤 page는 물리적 메모리에 존재, 어떠한 page는 물리적 메모리에 없음
  - 이 경우, 물리적 메모리로 들고 온다.
- 유효하지 않은 참조 (프로세스 주소 공간에 없는 주소 영역 참조 등)
  - abort / trap 처리
- obsolete, 오래된 데이터 갱신 (디스크에서 메모리로 들고온 데이터가 오래된 경우)
  - 디스크에 있는 원래 데이터가 더 최신 버전인 경우, 이는 1번째 경우로 취급해 처리함

< 기존의 valid - invalid >

- valid : process가 쓰고있는 page, context switch 마다 모두 invalid 초기화!
- invalid : 그 외 나머지

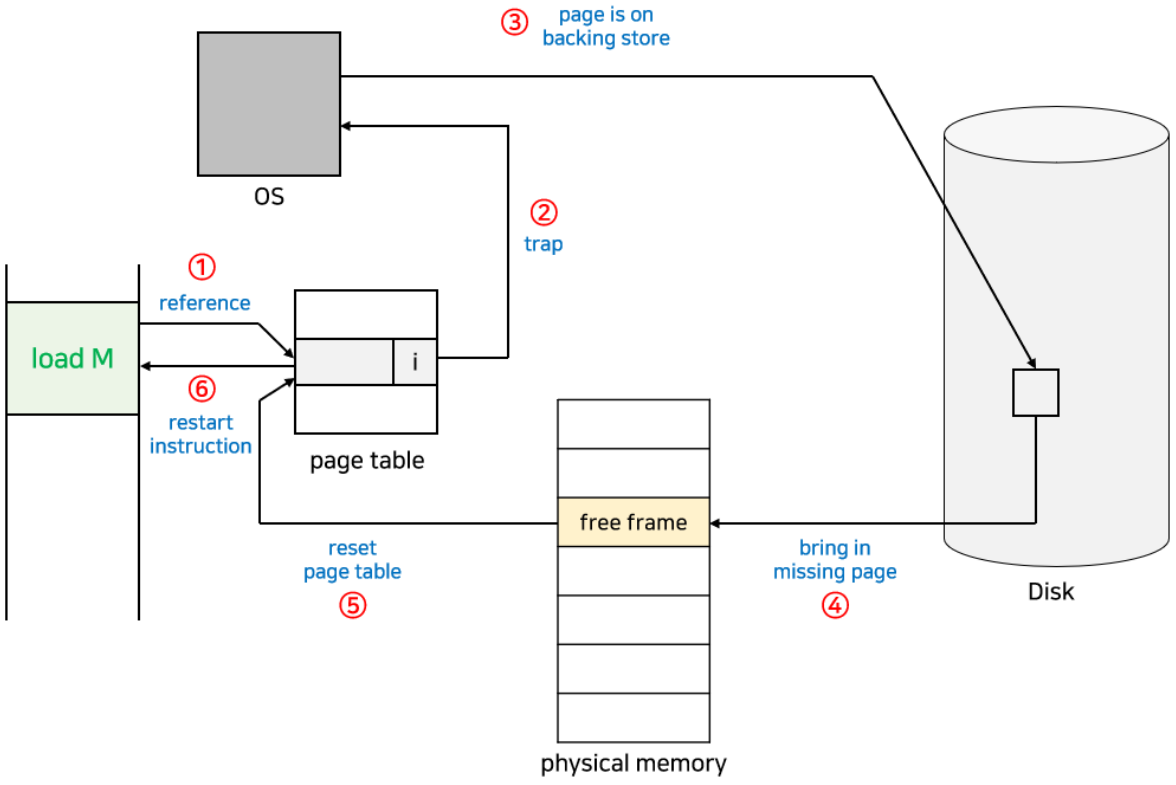
< demand paging의 valid - invalid >

- valid : 해당 page가 물리적 메모리에 존재하는 경우
- invalid : 메모리에 없거나, 메모리에 있긴 하되 오래된 버전인 경우

  ⇒ process가 context switch 되어도 따로 초기화 할 필요는 없어짐..

⇒ 처음에는 모든 page entry가 invalid로 초기화되어있고, 주소 변환 시 bit가 invalid로 되어있다면 page fault라는 오류가 발생

※ page fault



⇒ 유효하지 않은 page 접근시 HW(MMU)가 발생시키는 trap의 일종, page fault trap

⇒ trap 원인은 위의 3가지를 필두로 여러 가지가 있기 때문에, handler가 절차에 따라 적절히 처리!

⇒ 커널 모드로 전환 후, OS 내부에 있는 trap handler가 invoke 후, 다음과 같이 처리된다.

1. 일단, Invalid reference(bad address, protection violation) 일 경우 process 중단
2. 반면, not - in - memory라면, 그대로 다음 과정을 수행한다.
3. empty page frame(free frame)을 가져온다.

- 없으면, 이미 올라간 frame 중 하나를 쫓아내 빈 공간을 확보한다. (replace)

4. 해당 페이지를 disk 에서 memory로 읽어온다.
    a. disk I/O가 끝나기까지 이 프로세스는 CPU를 preempt 당함(wait or block)
    b. (단, 해당 프로세스는 OS 때문에, 타의적으로 block 당함)
    c. Disk read(I/O)가 끝나면 page table entry 기록, valid/invalid bit을 valid 로 바꿈
    d. ready queue에 block 당한 process를 삽입, 나중에 dispatch 하도록 한다.
5. page fault 처리가 끝나고, 다시 CPU가 ready 큐에 있던 해당 프로세스를 점유하게 한다.
6. 이 프로세스가 CPU를 잡고 다시 작동(running)시킨다.
7. 아까 중단 되었던 명령(instruction)을 재개한다.

※ 가상 메모리 & File 개념과 Page Fault

1. 운영체제가 데이터 파일의 일부를 Virtual Memory 주소공간 일부에 매핑을 해준다.
2. 프로그램이 실행되면서 이 메모리 위치를 접근했을 때 메모리에 없다면 Page Fault!
3. Page Fault가 일어났다면 운영체제는 Page Fault가 일어난 Page를 물리적 메모리에 올려준다.
4. 가상 메모리 Page가 물리적 메모리의 Page와 매핑 되어 접근할때는 운영체제 도움 X
    - 운영체제 도움없이 물리적 메모리에 읽거나 쓸 수 있다.
5. 메모리에 쫓겨날 때는 File System에 수정된 내용을 써주고 메모리에서 쫓아낸다.

※ page fault로 인한 단점?

⇒ Page fault Rate = 0 ≤ p ≤ 1.0 이라고 할 때,

- p = 0이면 no page fault이고 p = 1이면 매 참조마다 page fault이다.

⇒ 보통은 p가 매우 작은 편이기 때문에 page falut가 일어나지 않는다.

⇒ 그러나 한 번 page fault가 발생하면 엄청난 시간을 소비해야 한다.

```
Effective Access Time(EAT) =
   (1-p) × memory access
```

```
            + p × (OS&HW page fault overhead
          + [swap page out if needed]
          + swap page in
          + OS&HW restart overhead)
```

※ demand paging 파생형

- Pure demand paging (기본)
  - 참조할 때 까지 swap X, 참조 시에만 메모리 in
  - 메모리에 page 없는 상태로 프로그램 시작
- pre demand paging (파생)
  - 미리 page를 올려놓기
- read ahead (파생)
  - 연속적인 다음 몇 개의 page를 같이 가져오기

※ 참조의 지역성 (Locality of Reference)

- 프로그램의 어느 한 특정 작은 부분만 한동안 집중적으로 참조하는 현상
  - 페이지 참조 = 특정 시간에만, 매우 작은 페이지 집합에서만 발생
  - 이 덕분에, 성능 저하가 크지 않음, 많이 사용하는 page가 올라올 시 교체 X!

< 전체 주소 변환 과정 >

1. CPU는 TLB를 확인한다.
2. TLB hit인 경우 곧바로 주소를 변환하고, TLB miss인 경우 page table을 확인한다.
3. page table의 valid-invalid bit가 valid로 되어 있다면 주소를 변환하고 TLB에 page를 올린다.
   a. invalid라면 page fault가 발생한다.
   b. page fault가 발생하면 MMU가 운영체제에 Trap을 걸고 커널 모드로 들어감
   c. page fault handler가 invoke, 유효하지 않은 참조인 경우 프로세스를 종료
   d. 그렇지 않다면 빈 page frame을 얻는다.

e. 빈 frame이 없다면 메모리에서 victim page를 선택하여 대체한다.

4. 운영체제는 참조된 page를 디스크에서 메모리로 로드(I/O)한다.
5. disk I/O가 끝날 때까지 이 프로세스는 CPU를 빼앗긴다.
6. disk I/O가 끝나면 page table이 업데이트되고 valid-invalid bit가 valid로 바뀐다.
7. ready queue에 프로세스를 넣어준다.
8. 프로세스가 CPU를 잡게 되면 다시 이어서 수행한다.

---

3. Page Replacement

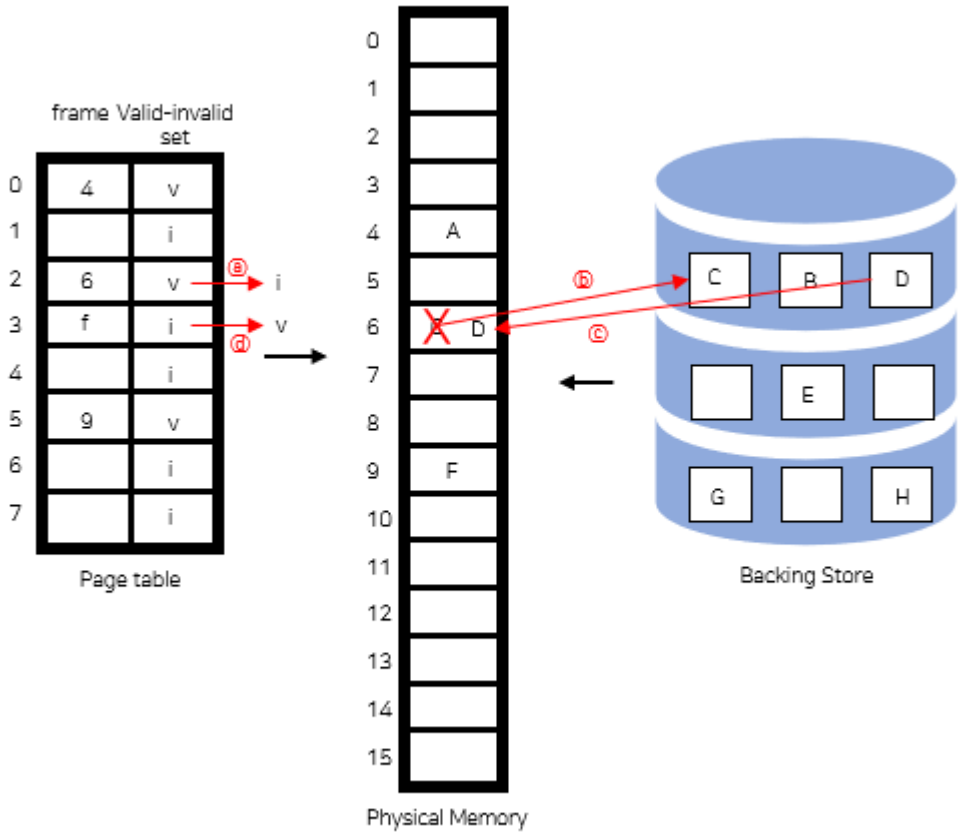
⇒ 페이지 교체를 포함하는 fault 처리 프로세스를 수정함으로써, 메모리의 과도한 할당을 방지

⇒ 페이지 swap의 오버헤드를 줄이기 위해서, modify bit를 사용!

- 디스크에 swap out된 수정된 pages들에 대한 수정 여부 판별
- 이를 통해, 가장 적게, 그리고 오래전에 사용한 page를 판별한다.

⇒ 페이지 교체는 논리적 - 물리적 메모리의 분리를 완성시킨다.

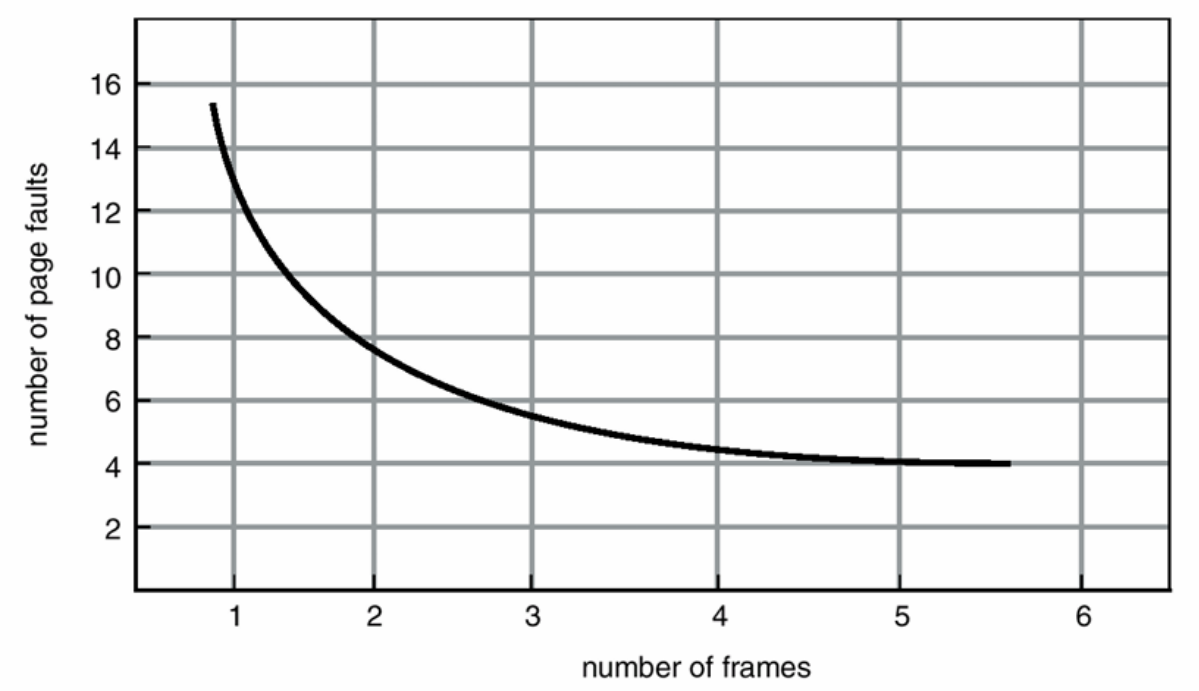
- 큰 가상메모리는 작은 가상메모리로 제공될 수 있다.



⇒ page frame이 존재하지 않는 경우에는 어떤 frame을 빼앗아올지 결정 요구 (free frame 선택)

⇒ 웬만하면 바로 사용되지 않은 page를 쫓아내는 것이 좋을 것

⇒ 동일한 페이지가 여러 번 메모리에서 쫓겨났다가 다시 들어올 수 있다.



⇒ 기본적으로 page fault rate를 최소화하는 것이 목표

- 이에 대한 알고리즘을 Replacement Algorithm
- 성능 평가 : 주어진 page reference string에 대해 page fault를 얼마나 내는지를 조사
- page reference string은 참조되는 일련의 page 번호

※ 기본 페이지 교체 루틴

1. 디스크에서 불러올 페이지 (위치)를 찾는다.
2. Free Frame(메인 메모리 내 빈 공간)을 찾는다.
    a. 만약 Free Frame이 있다면, 그것을 사용한다.
    b. 없다면, victim frame을 선택하기 위한 페이지 교체 알고리즘을 사용한다.
3. free frame에 설정된 page를 가져온다. free frame table과 page를 갱신한다.
4. 프로세스를 재시동한다.

※ pre - replacement(사전 대치) & pre - paging(사전 적재)

⇒ 페이지 부재의 발생 → 큰 오버헤드 발생!

1. 페이지 부재 발생 시 페이지 부재 트랩에 의해 처리

2. 이후에 해당 프로세스를 다시 실행시키려면 스케줄링이 실행
3. 만약 페이지 대치까지 수행되어야 하는 상황이라면 여기서 더 많은 시간이 소요

⇒ 이를 해결하기 위해서는 페이지 대치 알고리즘을 사전에 실행시키는 방법!

⇒ 항상 빈 프레임을 가지고 있다면 ?

- 페이지 부재 발생 시 페이지 대치 알고리즘까지 수행해야 하는 것을 미리 방지 가능

⇒ 미리 선정된 희생 page를 디스크로 내보내는 출력 작업(write-back)은 CPU와 병행 가능하기

때문에 더 효과적임

(단, 현재 입출력에 사용되는 페이지는 대치되지 말아야 하는 조건 필요)

⇒ 더 나아가, 페이지 부재 발생 시점에 페이지를 적재하는 방법이 아닌, 미리 여러 페이지를 한번에 적재해놓는 방법도 존재

(페이지 참조에 대한 예측이 필요하지만 다양한 정보를 활용하는 것을 통해 구현 가능)

ex. 서브루틴(함수), 행렬이 참조되는 경우 연관 페이지 모두를 적재하는 방법

→ 컴파일러에 의해 필요 정보가 제공될 수 있음

※ Paging size

⇒ 64비트에서는 더 큰 페이지 사이즈를 사용하게 됨

⇒ size ↑, page 개수 ↓, page table size ↓ → I/O 효율 증가, 지역성이 하나의 page로 처리 가능

(page fault 감소로 지역성 향상)

⇒ 페이지 크기 키우면, Disk transfer의 효율성 증가!

- Disk 자체는 Seek와 rotation을 통해 작동
- 디스크 헤드가 이동하는 동작인 Seek의 작동 시간이 굉장히 김
- 최대한 많은 양을 올리는 것이 Disk transfer의 효율성에 더 좋다.

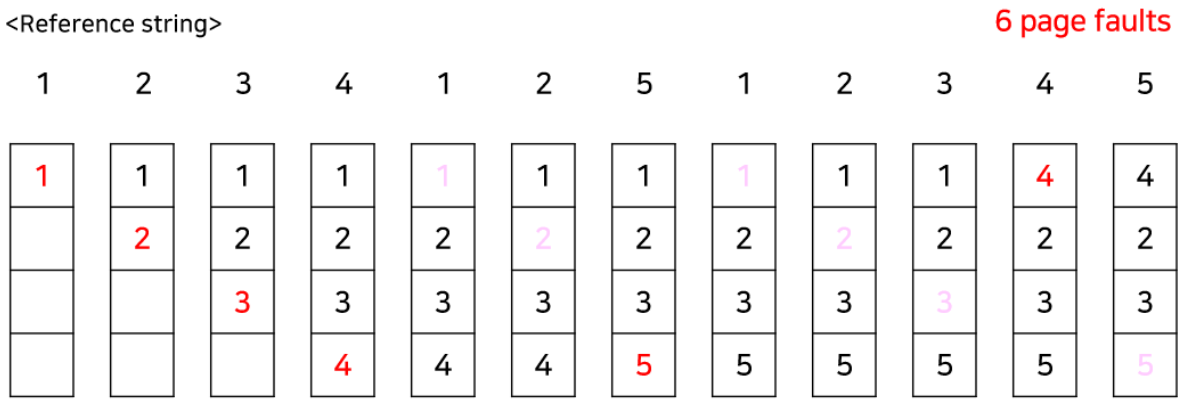
⇒ TLB Reach = (TLB size * Page Size), TLB에서 메모리 접근의 총합, 클수록 이득이다.

- TLB Size는 하드웨어적으로 고정, page Size가 클수록 유리

⇒ 그러나, 내부 단편화 문제 발생도 존재할 수 있음

⇒ 이러한 장점 때문에, 현재 트렌드는 Page Size를 키우면서 Multiple Page Size를 제공!

1. Optimal Reference (이론적 최상의 결과 도출)



- 빨간색 숫자 : page fault가 발생한 경우 → 맨 처음 page에 접근하거나, 메모리에 없는 경우
- 분홍색은 frame에 이미 존재해서 Hit된 경우 & page fault가 발생하지 않은 경우
- Optimal algorithm은 항상 최적의 해 도출 ⇒ 위 예시 이외에 6번 미만의 fault은 불가능하다.

⇒ Optimal Algorithm은 가장 먼 미래에 참조되는 page를 대체하는 방법

⇒ page falut를 가장 적게 가져가는 방법이고 항상 최적의 결과를 가짐

⇒ 다만, 미래의 참조를 모두 알고 있어야만 가능 → 실제 사용 불가능

⇒ 이 방식은 다른 알고리즘의 성능에 대한 상한선 upper bound을 제공하는 역할 (기준)

2. FIFO Algorithm

<Reference string>

1	2	3	4	1	2	5	1	2	3	4	5

9 page faults

1	1	1	4	4	4	5	5	5	5	5	5
	2	2	2	1	1	1	1	1	3	3	3
		3	3	3	2	2	2	2	2	4	4

10 page faults

1	1	1	1	1	1	5	5	5	5	4	4
	2	2	2	2	2	2	1	1	1	1	5
		3	3	3	3	3	3	2	2	2	2
			4	4	4	4	4	4	3	3	3

⇒ 제일 먼저 들어온 것을 먼저 쫓아내는 방법, 미래를 모르는 경우에도 사용할 수 있다.

⇒ 모든 page가 평등하게 frame에 존재하며, 구현하기 쉽다는 장점

⇒ 중요한 특정 page는 항상 필요할 수 있는데, 그런 경우에도 쫓아내버린다는 단점

Belady's anomaly : frame이 늘어나도 page fault가 감소하지 않고 오히려 늘어나는 경우가 존재

3. LRU Algorithm (Least Recently Used)

<Reference string>

8 page faults

1	2	3	4	1	2	5	1	2	3	4	5
1	1	1	1	1	1	1	1	1	1	1	5
	2	2	2	2	2	2	2	2	2	2	2
		3	3	3	3	5	5	5	5	4	4
			4	4	4	4	4	4	3	3	3

⇒ 가장 오래전에 참조된 페이지를 지우는 방법, 또한 Optimal에 가장 근접한 방법

- 각 페이지마다의 타임스탬프가 요구됨, 추가 메모리 (page table) 트래픽
- 또한, 타임스탬프가 가장 작은 페이지를 찾는 로직이 추가적으로 필요!
- 너무 큰 공간/시간 오버헤드 → 커널에 통합 X
- 그래서, 구현하기 어렵고 접근 빈도를 고려하지 않는다는 단점

⇒ 1→2→3→4→5→6→... 처럼, 1개만 참조하는 경우에 최악의 성능을 보인다.

⇒ FIFO와는 다르게 Belady's anomaly가 발생하지 않는다.

< LRU Approximate Algorithm, 근사 >

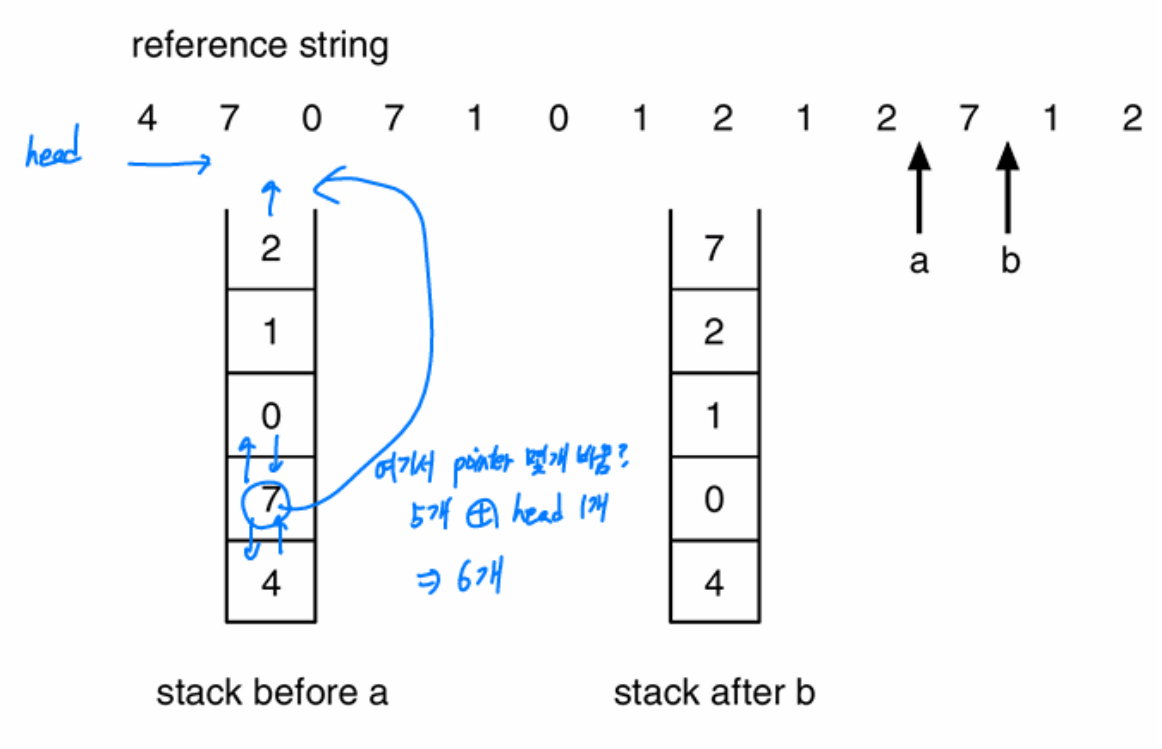
⇒ 각 page 마다의 timestamp → 오버헤드 문제, 이를 줄이고자 고안된 기법 (단, 성능도 낮아짐)

(정확한 timestamp 대신 상대적인 순서만 기억하자는 취지)

※ Counter 구현 (Memory Reference 횟수를 나타냄)

- 모든 page는 Counter를 가짐, 매번 메모리 참조 시마다 1씩 증가시켜 나간다. (Logical clock)
- CPU에 CLOCK을 둬서, 그것을 그대로 복사하여 참조 될 page에 그대로 기록!
- 페이지 교체 로직 중에, 가장 작은 카운터를 page table에서 찾는다.
  - 매번 메모리 접근마다 Counter write time ⇒ 추가 오버헤드
  - 매번 교체 작업마다의 탐색 오버헤드
  - Count 오버헤드 등...

※ Stack 구현 (Memory Reference 횟수를 나타냄)



- double link 형태로 page stack 형태를 유지
- 교체를 위한 탐색은 필요하지 않다.
- 참조된 페이지 A에 대해,
  - Stack의 맨 꼭대기로 옮긴다.
  - (스택 꼭대기로의 포인터 포함) 6개의 포인터 갱신 수행 ⇒ 오버헤드

※ Reference bit (LRU 근사 알고리즘의 기본 토대)

⇒ 각 페이지에 비트가 연결될 때 처음에는 = 0, 페이지가 참조될 때 마다 1로 갱신

⇒ 주기적으로 0으로 바꿈으로써, 다음 주기가 오기 전에 1로 바뀌면 그건 최근에 참조됨을 의미

⇒ 전체 페이지 사용 순서는 모르나, 주기동안 어떤 페이지가 사용되었는지 알 수 있음!

⇒ 여전히 page 참조 순서를 예측할 수 없기 때문에, 주기 설정이 어려움!

※ Additional-Reference-Bits Algorithm

⇒ 8개의 추가 비트로 history 역할을 수행함

- 참조 비트는 주기적으로 8개 중 가장 높은 차수 비트로 이동 (주기 2면, 1번째 → 3번째 → ...)

- 주기 : 타이머 인터럽트, 참조 바이트를 우측으로 1bit shift
- 참조 비트를 맨 왼쪽 최상위 비트 MSB에 복사
- 참조 바이트 값이 가장 작은 페이지가 참조된지 가장 오래됨 → 해당 페이지를 선택하여 교체

⇒ Ex) 00000000 (한번도 참조 안한 페이지) → 10101010 (2주기마다 1번씩 참조한 경우)

⇒ Ex) 00000000 → 11111111 (최근 8회 인터럽트, 주기동안 각 구간마다 1번 이상 참조한 경우)

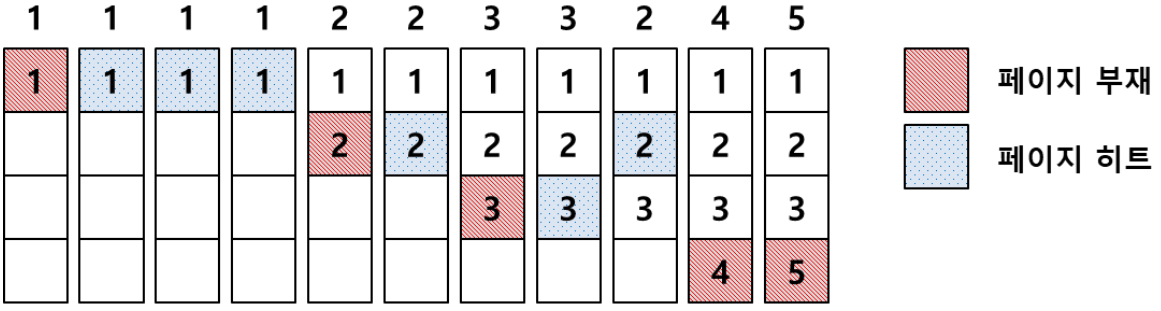
⇒ 이를 통해, 참조 빈도를 비교가능하여 정확도 ↑

⇒ 그러나, 8bit 사용 & 최솟값을 찾는 알고리즘 → overhead ↑

⇒ 연결 리스트로 LRU를 구현하면 O(1)만에 page를 탐색하고 삽입

⇒ 제일 최근에 참조된 page를 가장 앞으로 옮기는 방식으로 연결 리스트를 구현하게 된다면, replace가 일어날 때 가장 뒤에 있는 page를 바꿔주면 된다.

4. LFU Algorithm (Least Frequently Used)



⇒ 참조 횟수가 가장 적은 page를 지우는 방법

⇒ LRU에 비해 장기적인 시간 규모를 보기 때문에 page의 인기도를 조금 더 정확히 반영하는 장점

⇒ 최근성은 반영하지 못하지만, LRU보다 구현이 복잡하다는 단점

⇒ 최저 참조 횟수인 page가 2개 이상인 경우에는 LFU 알고리즘 자체에서는 임의로 page를 선정

⇒ LFU 알고리즘은 성능 향상을 위해 가장 오래전에 참조된 page를 지우는 식으로 구현

⇒ LFU를 LRU처럼 연결 리스트를 이용해서 구현!

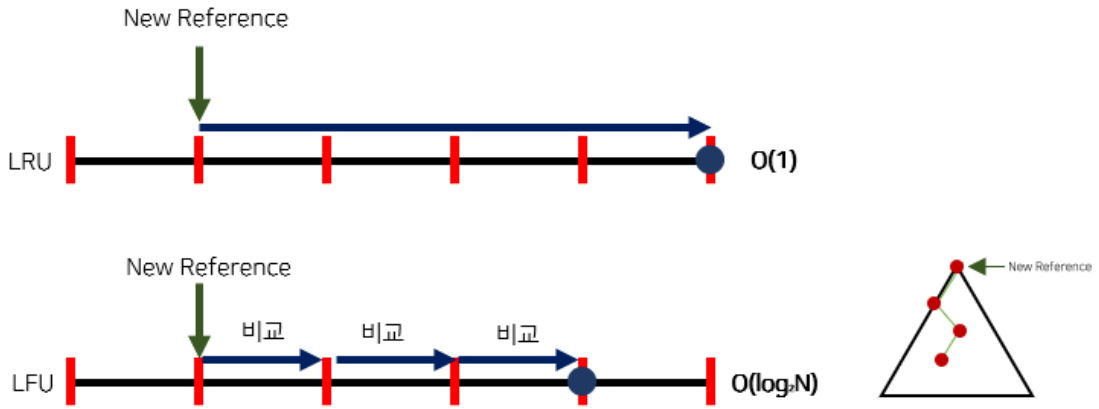
- 쫓아내야할 page를 찾는데 O(n)의 시간이 걸리게 되어 느리다.

⇒ 반면, 힙(heap)을 사용하면 최소 빈도를 갖는 page를 찾거나 삽입!

- 삭제하는데 O(logn)의 시간이 걸리게 되므로 훨씬 효율적으로 구현

⇒ 참조 바이트 값이 가장 작은 페이지가 참조된지 가장 오래된 것 → 해당 페이지를 선택하여 교체

※ LRU vs LFU



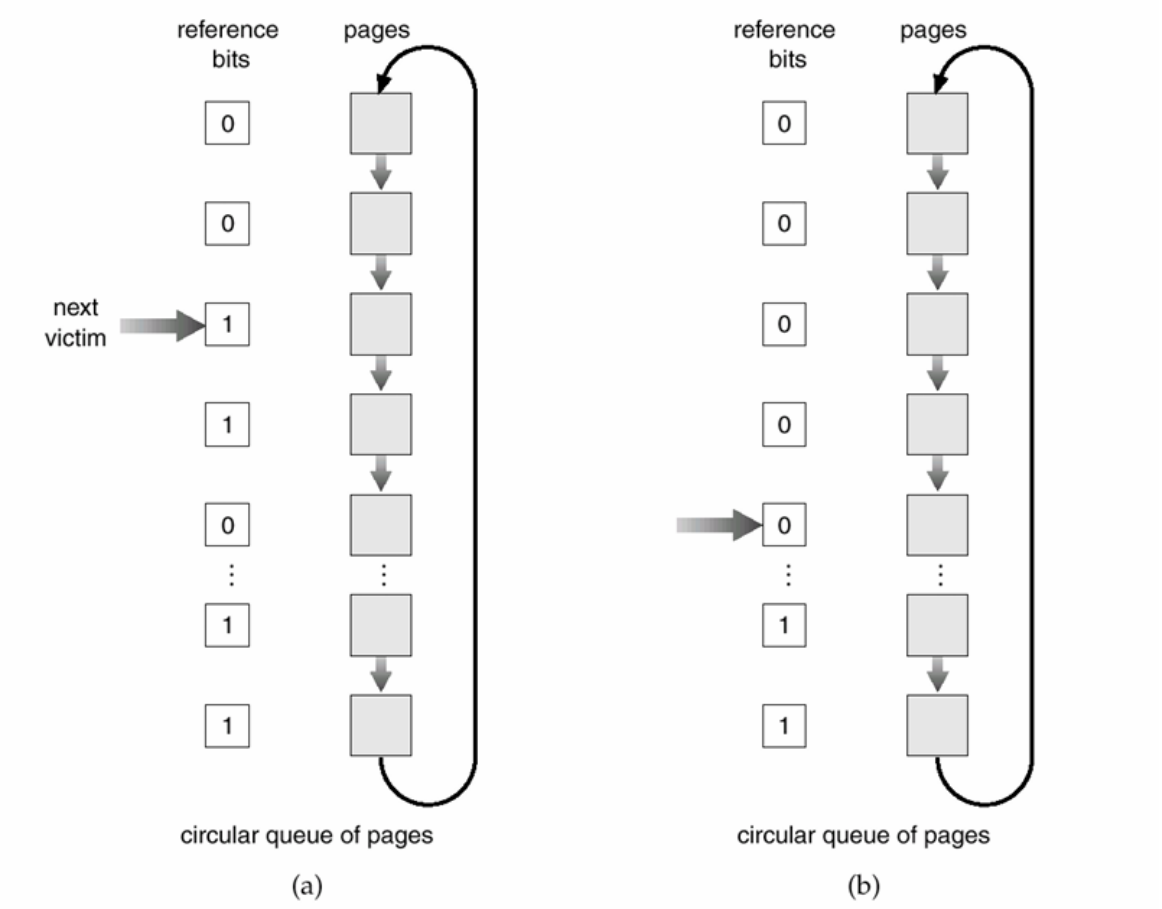
⇒ LRU는 방금 참조된 페이지가 가장 뒤로 가는 방식이고 연결리스트를 통해 구현한다.

⇒ LFU는 참조 횟수를 비교하고, 힙을 이용해 구현하게 된다.

⇒ 힙은 최소힙 → 많이 참조된 페이지일수록 아래쪽에 있고, 쫓아낼 때는 Root를 쫓아내고 재구성

⇒ 캐싱 기법 중 MRU는 LRU 알고리즘을, MFU는 LFU 알고리즘을 사용

5. Second chance (clock) algorithm



⇒ Paging System에 LFU와 LRU를 적용할 수 없기 때문에 이에 대한 대안으로 나온 알고리즘

(NUR(Not Used Recently) 또는 NRU(Not Recently Used) 라고도 불림)

⇒ Clock Algorithm은 LRU의 근사(approximation) 알고리즘 → 최근 참조 의미하는 Reference bit

⇒ 앞서 언급한 참조 비트(Reference bit) 필요, 원형 Queue처럼 관리 수행

⇒ (참조를 한 번도 안 한) 참조 비트 0을 찾을 때까지, 다음 page를 가리키는 포인터를 이동한다.

- 해당 참조 비트가 0이면 그대로 해당 프레임의 페이지를 교체

⇒ (clock order 로) 만약 교체할 page의 참조 비트가 1을 가질 때, 다음 과정을 처리한다.

(즉, 참조가 됐을 때 해당 page 참조 비트를 0으로 처리 후, 다음 FIFO 페이지 → pointer 전진)

(이 과정은 해당 포인터가 가리키는 페이지 참조비트가 1일 때 수행)

- 참조 비트를 0으로 처리하고, 다음 기회에도 0이라면 메모리 안에 있는 page 교체
- 다음 기회에 해당 위치에 왔을 때 또 1로 설정되어 있다면, 자주 사용되는 page라는 의미!

⇒ 참조 비트가 1에서 0으로 변경된 페이지는 다른 모든 페이지들이 교체되거나 기회를 받을 때까지 교체되지 않는다.

⇒ 따라서 참조 비트가 계속 설정되어 있을 정도로 자주 사용되는 페이지는 전혀 교체되지 않을 것이다.

※ Enhanced Second Chance Algorithm

⇒ 위 알고리즘을 조금 더 개선한 방식

⇒ 최근에 해당 page가 변경이 되었는지를 나타내는 Modified bit(dirty bit)가 추가

⇒ Modified bit가 1이라면 page가 변경되었기 때문에 교체를 하면 디스크에 해당 내용을 반영

⇒ I/O 작업이 동반되므로 시간이 오래 걸린다.

- 따라서 "Reference bit == 0? → Modified bit == 0?" 순서로 우선순위

---

4. Caching

⇒ 캐싱은 비용이 비싸고 용량이 적은 하드웨어(=캐쉬)에 요청된 데이터를 저장해두었다가, 후속 요청 시 캐쉬로부터 직접 서비스하는 방식이다.

⇒ paging system 외에도 cache memory, buffer caching, Web caching 등 다양한 분야에서 사용

⇒ 교체 알고리즘에서 삭제할 항목을 결정하는 일이 지나치게 많은 시간이 걸리는 경우, 실제 캐싱 시스템에서 사용할 수 없다.

⇒ 단, 디스크 캐시 등의 다른 캐시 분야에선 아래 알고리즘을 활용 가능 하다.

※ LRU(Least Recently Used)

- 시간을 기준으로 해서 가장 오래 전에 썼던 것을 지우는 알고리즘
- 스택, 연결 리스트 등으로 활용, 대부분의 운영체제가 채택한 방식

※ LFU(Least Frequently Used)

- 가장 횟수가 적게 액세스 되었던 것을 지우는 알고리즘 (블록마다 참조 횟수 counter 활용)

※ 캐싱에선 왜 LRU 및 LFU를 그대로 활용할 수 없는가?

1. Process A의 논리 메모리에서 매 순간마다 instruction을 하나씩 읽어서 실행
2. page table을 이용해 논리적인 메모리를 물리적인 메모리로 변환, 이를 얻어 CPU에 전달
3. 만일 요청한 페이지가 메모리에 올라가 있다면 운영체제가 아닌 하드웨어의 동작만으로 작동
4. 반면 invalid 상태라면(해당 정보가 물리적 메모리에 올라가있지 않다면) page falut가 발생
5. 이제 CPU의 제어권이 운영체제에 넘어가게 되고, 메모리에 올라가 있는 것을 쫓아내어, 요청 페이지를 메모리에 올림
6. page fault일 때만 운영체제는 CPU 제어권을 가짐
   a. 해당 페이지가 메모리에 올라와 있다면 운영체제는 주요 정보인 **참조횟수**를 알 수 없음
   b. 이 때문에, CPU 접근 시간을 파악할 수 없음
7. 그렇기 때문에 Virtual memory system(Paging System)에서는 LRU와 LFU를 적용할 수 없다.

---

5. Page Frame Allocation

⇒ 얼마만큼의 frame을 프로세스에 할당하는지에 관한 문제

⇒ 메모리 참조 명령어 수행시 명령어, 데이터 등 여러 페이지를 동시에 참조!

- 최소한 할당되어야 하는 frame의 수가 정해져 있다.
- HW 적으론, 하나의 명령어 처리를 위해 대략 6개 page 필요!
  - 명령어가 담긴 Page
  - 명령어 상의 주소가 참조하는 Page
  - 그 참조된 주소의 내용이 가리키는 Page

    - inst는 2개의 page에 걸쳐 존재 & “참조” 처리에 2개 & “가리킴” 처리에 2개
- loop를 구성하는 page들은 한 번에 하지 않는다면 매 loop마다 page fault가 발생
    - 연관 관계에 있는 페이지 집합은 한꺼번에 할당(allocate)해줘야 한다.

⇒ 보통은, 사용할 수 있는 frame의 총 양보다 넘어서거나, 또는 최소한의 필요한 frame보다는 적게 할당하면 안 된다.

⇒ page frame을 할당하는 방식?

- 프로세스마다 균일하게 할당 (Equal allocation)
    - 말 그대로 frame 수, 프로세스 수만큼 각 프로세스마다 frame을 할당
- 특정 기준에 따라 할당
    - Proportional allocation : 프로세스 크기에 비례하여 할당하는 것
    - Priority allocation : 프로세스의 priority(빨리 끝나는 순 등)에 따라 다르게 할당

※ 프로세스가 page fault를 발생시켰을 때 대체될 frame의 그룹에 따라 2가지로 나뉜다.

1. **Global Replacement**

    ⇒ replace할 때 다른 process에 할당된 frame을 빼앗아올 수 있다.

    ⇒ Process별 할당량을 조절하는 방법이지만 자신의 page fault rate를 조절할 수 없다.

    ⇒ 우선 순위가 높은 프로세스에게 작은 프로세스를 희생하면서 할당된 프레임 수를 늘린다.

    ⇒ Working set PFF 알고리즘 사용

    ⇒ 좋은 처리량을 가지고 있어 현재는 대부분 Global Replacement 를 사용

2. **Local replacement**

    ⇒ 자신에게 할당된 frame 내에서만 교체(replacement)하는 방법

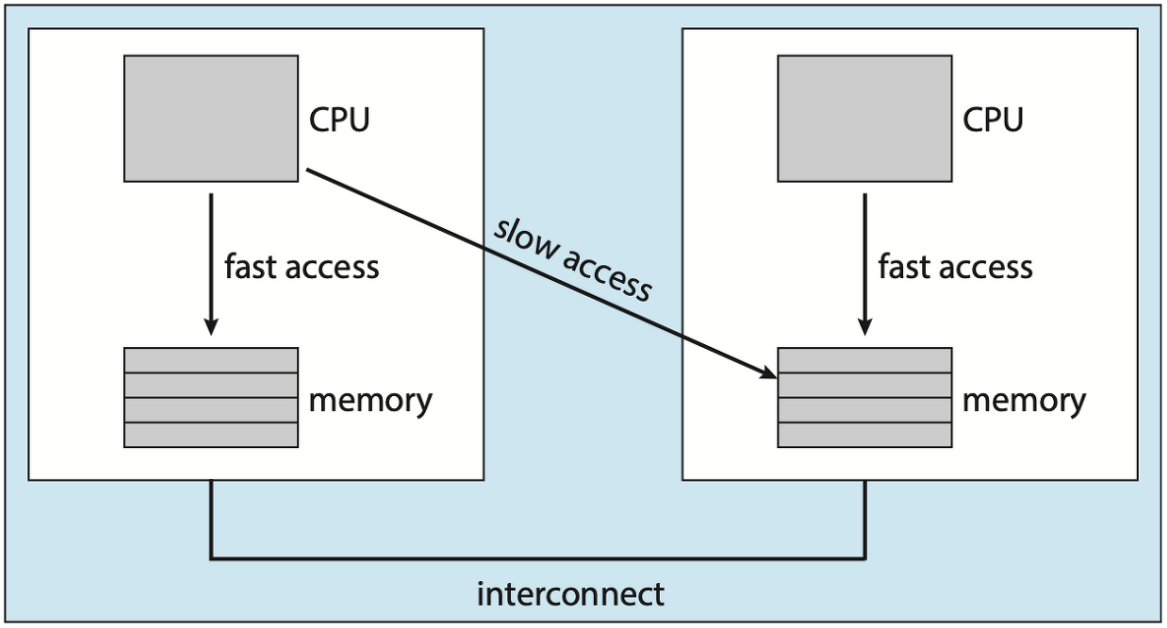
    ⇒ FIFO, LRU, LFU 등의 알고리즘을 procees별로 운영시 사용한다.

    ⇒ 알고리즘을 프로세스마다 독자적으로 운영하는 경우 가능하다.

    ⇒ 쉬고 있는 메모리를 사용할 수 없기 때문에 비교적 비효율적이다.

< 심화 >

1. 비균등 메모리 접근 (Non-Uniform Memory Access, NUMA)



⇒ 비균등 메모리 접근 시스템(NUMA)은 복수의 CPU를 가진 시스템(다중 처리기 시스템)

⇒ 특정 보드 상의 CPU는 같은 보드의 메모리를 다른 보드의 메모리보다 더 빠르게 접근

⇒ 프로세스가 현재 실행 중인 CPU에 가능한 가장 가까운 메모리 프레임이 할당

- 스케줄링 시스템도 수정이 필요!
    - 스케줄러가 프로세스가 마지막으로 실행된 CPU를 추적할 수 있게끔 수정
    - 각 프로세스를 직전에 실행된 CPU에 스케줄
    - 메모리 관리 시스템은 스케줄 된 CPU와 가까운 프레임 할당

※ 어느 페이지를 어느 프레임에 할당하느냐에 따라 성능에 커다란 영향!

⇒ 이를 통해 캐시 적중률은 상승하고 메모리 접근 시간은 감소!

2. Copy-on-Write (쓰기 시 복사)

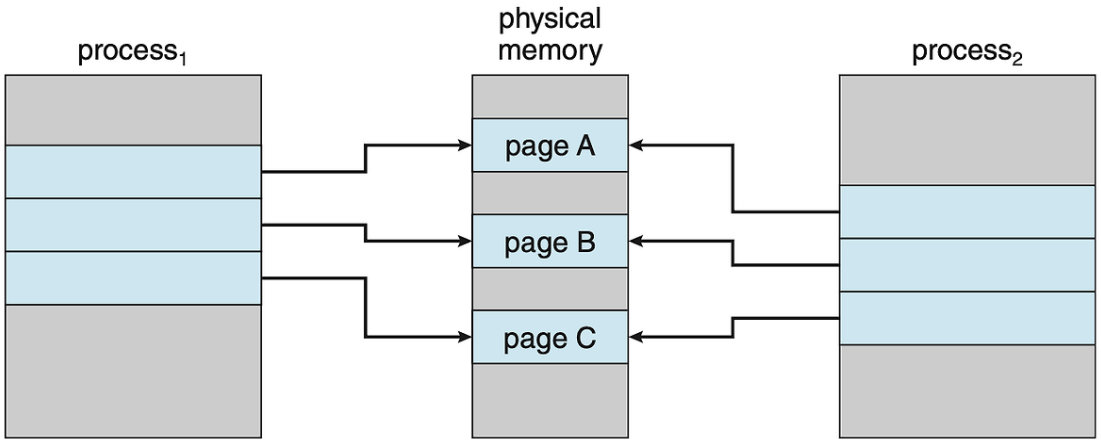
⇒ fork() 시 자식 프로세스를 위해 부모 프로세스의 프레임들을 복제

⇒ 대부분의 자식들은 바로 exec() 시스템 호출 → 부모로부터 복사한 페이지들은 쓸모가 없어진다.

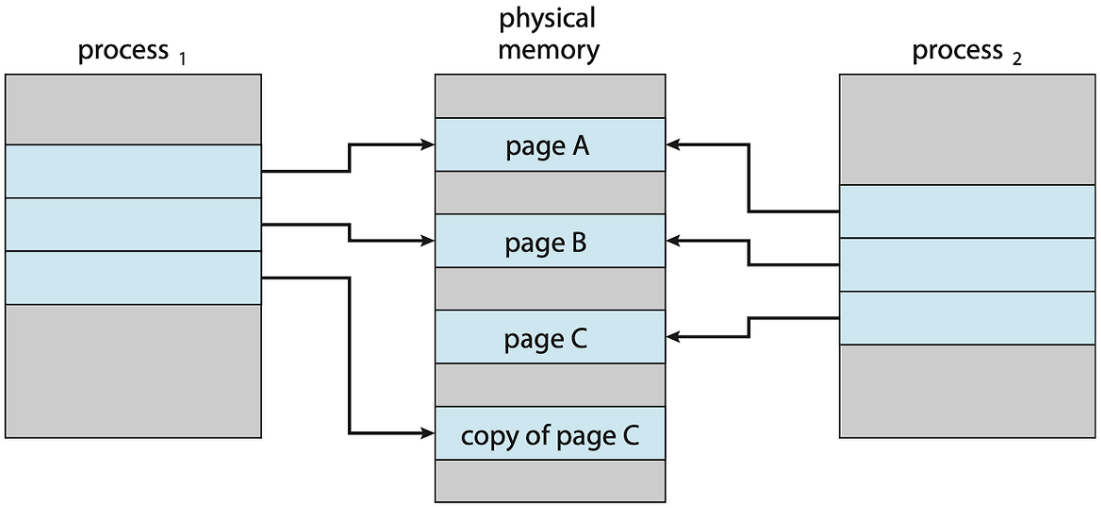
⇒ 따라서 부모 페이지를 전부 복사해오는 대신, 쓰기 시 복사 방식을 사용하여 낭비 ↓

1. 자식 프로세스 시작 시 부모의 페이지를 공유하여 당분간 함께 사용하도록 한다.

   (이 페이지는 copy-on-write 페이지로 마크를 한다.)



2. 만약 둘 중 한 프로세스가 **공유 중인 페이지에 쓰기를 했을 경우, 그때 새로운 프레임을 생성**하여 그 페이지의 복사본이 만들어진다.



⇒ 페이지 복사본을 만들 때 빈 페이지는 어떻게 할당될까?

⇒ 빈 페이지가 프로세스에게 할당되는 경우는 아래의 2가지가 존재한다.

1. Copy-on-write을 진행할 때
2. 스택, 힙 공간을 확장해야할 때

⇒ 이때 미리 마련된 **빈 페이지 집합(pool)**에서 할당 가능한 페이지를 찾아, **zero-fill-on-demand**

기법으로 할당해주게 된다.

- zero-fill-on-demand 기법 : 페이지의 내용을 다 0으로 채워 이전 내용을 지움 (초기화)

3. Memory-Mapped Files

⇒ open(), read(), write() 시스템 호출을 사용하여 디스크에 있는 파일을 순차적으로 읽는 기존 방식

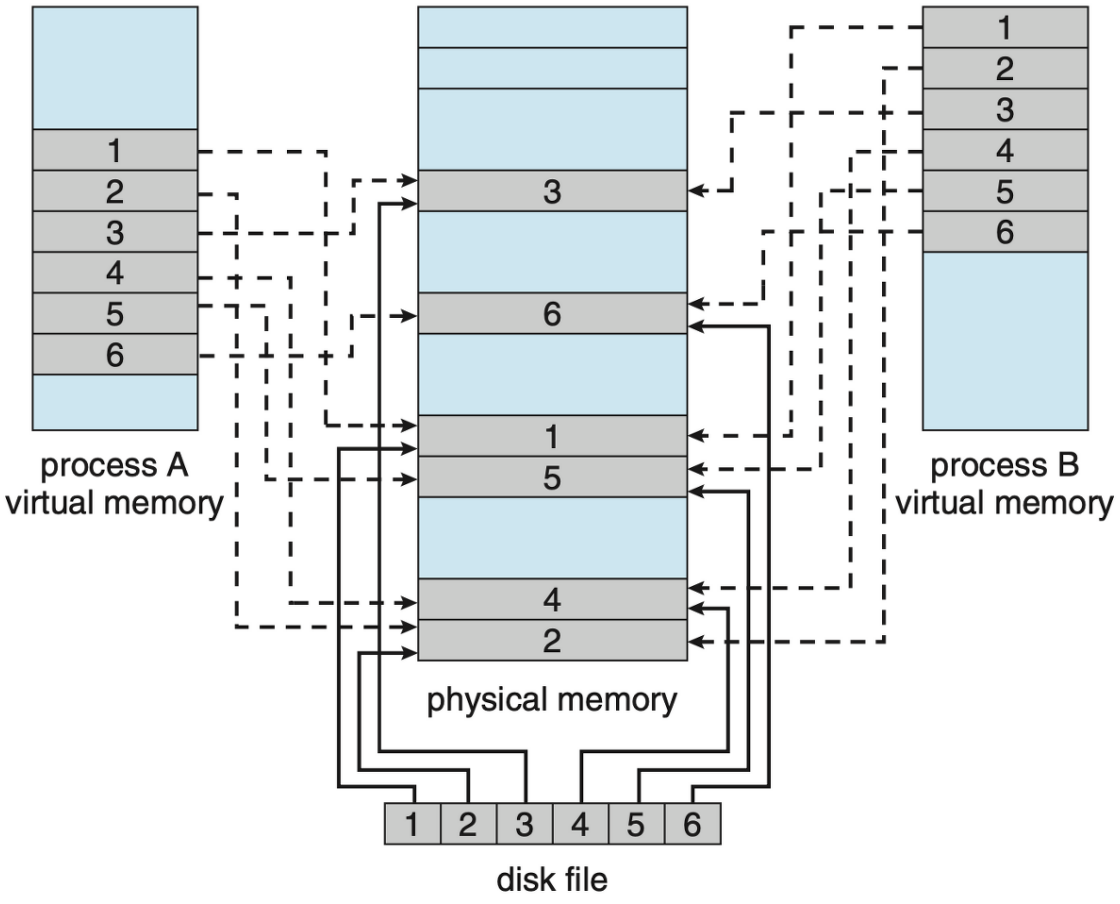
⇒ 파일이 매번 액세스될 때마다 시스템 호출을 한 다음에 디스크를 접근, 성능이 안좋음

⇒ 시스템 콜 대신, 디스크 입출력을 가상 메모리 기법을 사용하여 메모리 참조 방식 수행

< 메모리 사상(memory-mapping) 파일 >

⇒ 프로세스의 가상 주소 공간 중 일부를 관련된 파일에 할애

- 시스템 콜 대신 메모리를 접근하도록 하는 방식



⇒ 파일의 메모리 매핑은 프로세스의 페이지 중 일부분을 디스크에 있는 파일의 블록에 매핑하는 것을 통해 이루어진다.

⇒ 파일에 대한 첫번째 접근은 일반적인 요구 페이징 과정에 따라 페이지 부재를 발생

⇒ 파일의 내용 중 페이지 크기만큼의 해당 부분이 파일 시스템으로부터 메모리 페이지로 읽음

⇒ 이후의 파일 쓰기와 읽기는 일반적인 메모리 액세스와 같이 처리

※ 장점

1. read()와 write()를 사용하는 오버헤드 없이 파일을 메모리를 통해 조작할 수 있고, 더 단순하고 더 빠르게 파일을 접근하고 사용할 수 있어 성능이 향상된다.

2. 여러 프로세스가 메모리 사상 파일을 통해 **특정 파일을 공유**하는 것도 가능하다.
    a. 이때 copy-on-write 기능을 지원하여 파일을 read-only로 공유한다고 가정

b. 어느 한 프로세스가 해당 파일을 수정하기 시작한다.

c. 그 시점에 별도의 페이지 복사본을 만들어 준다.

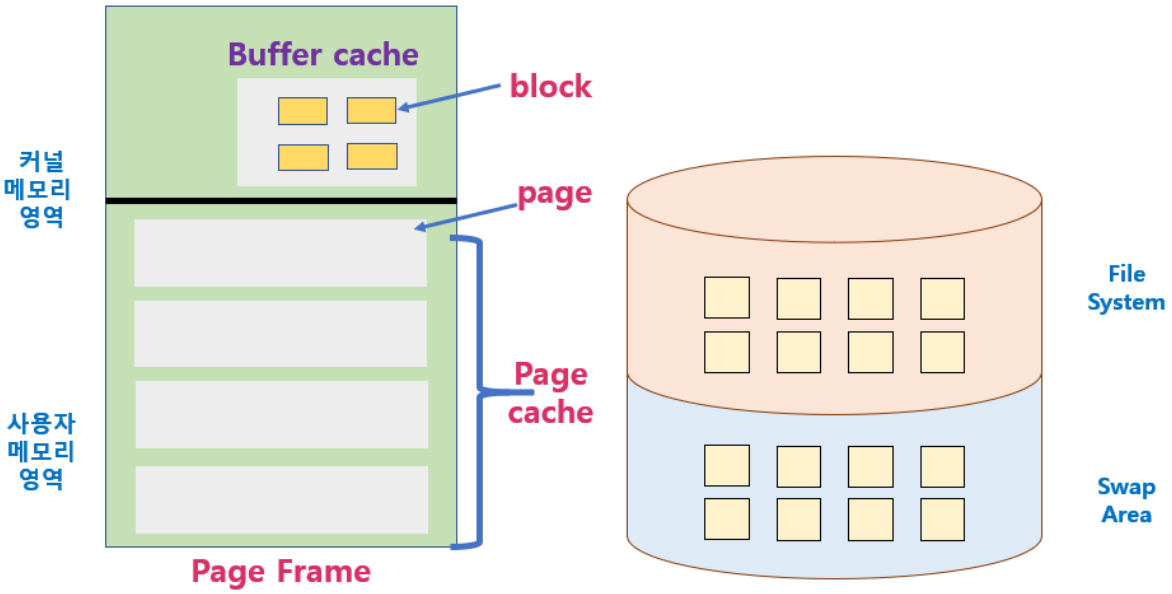
3. 메모리 매핑 파일을 이용하여 **공유 메모리**도 구현 가능하다.

   a. 동일 파일을 프로세스들의 가상 주소 공간에 매핑

      - 프로세스들은 공유 메모리를 통해 통신을 할 수 있다.

   b. 매핑된 파일은 통신하는 프로세스들 사이의 공유 메모리 영역으로 동작한다.

※ Page Cache & Buffer Cache



1. Page cache

⇒ Virtual Memory의 Paging System에서 사용하는 page frame을 caching의 관점에서 설명하는 용어

⇒ Page Frame에다가 당장 필요한 내용을 메모리에 올려두고 필요하지 않은 내용들을 쫓아낸다.

⇒ Memory-Mapped I/O를 쓰는 경우 file의 I/O에서도 page cache를 사용한다. 페이지 단위는 4KB

※ Memory-Mapped I/O

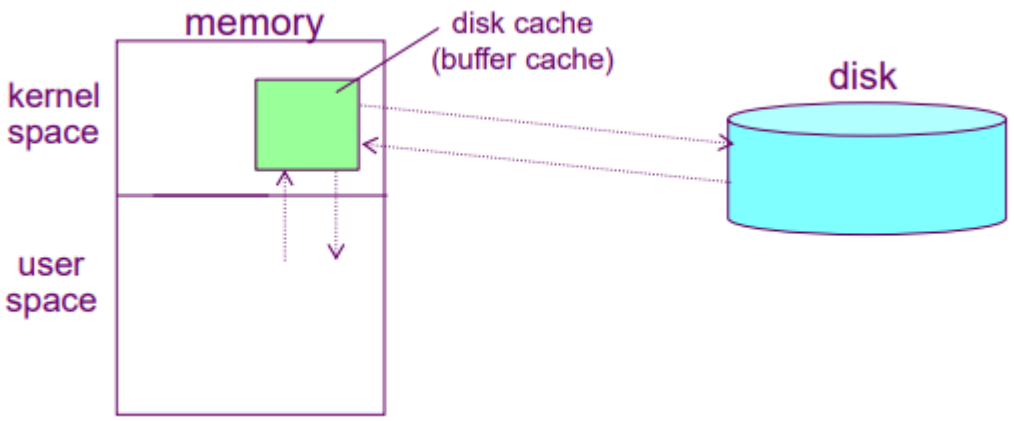
⇒ 파일 입출력을 read/write sys call을 통해 작업하지 않고, File의 일부를 가상 메모리에 mapping

⇒ 매핑시킨 영역에 대한 메모리 접근 연산을 통해, 파일의 입출력 수행이 가능

⇒ 이미 메모리에 올라온 내용에 대해서는 운영체제를 호출 X!

- 자신의 메모리에 접근하듯이 File 자체를 읽고 쓸 수 있음

2. Buffer Cache



⇒ 파일 시스템을 통한 I/O연산에서 사용되는 저장공간

⇒ 디스크라는 저속 저장 장치를 위해, 메인 메모리라는 고속 저장 장치에 할당한 저장 공간

⇒ 커널이 사용자의 요청을 받아서 디스크에서 가져온 정보를 사용자에게 바로 주지 않고, Buffer cache에 저장한 뒤에 전달

- 디스크에서 가져온 블록을 사용자에게 줌과 동시에 디스크 캐시에도 저장

⇒ 이후에 버퍼에 저장된 내용을 요청하면, 다시 디스크까지 가지 않고 버퍼에서 데이터를 전달

- File 사용의 지역 locality 활용
- 없을 경우에만, 다시 디스크로 가서 읽어 들여온다.

⇒ 모든 프로세스가 공용으로 사용하며, LRU와 LFU와 같은 Replacement algorithm이 필요

< Buffer Cache 필요성 >

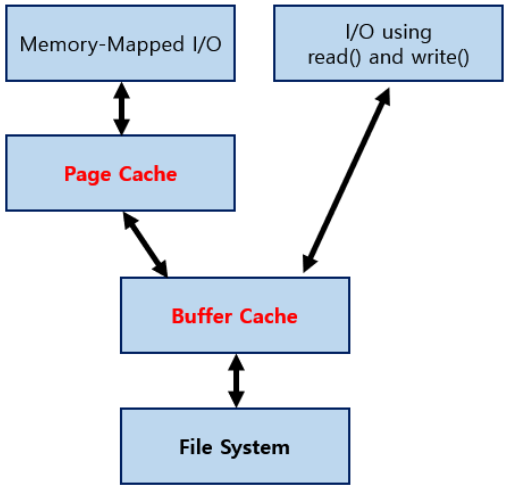
1. User Process A, B, C가 실행 중이라 가정한다. 이 때 A가 디스크 내의 어떤 블록 read 요청!
2. 요청은 시스템 콜로 운영체제에게 전달 → 커널 I/O 서브 시스템이 받음
   a. 서브 시스템이 디바이스 드라이버로 전송, 디스크에게 명령을 내린다
3. 프로세스 스위칭 → 다른 프로세스로 제어가 넘어가 실행
   a. 언젠가 요청된 블록이 디스크에서 읽힘 → 인터럽트 실행
   b. 인터럽트 처리 절차 등 복잡한 과정을 거쳐 읽혀진 블록이 프로세스 A에게 전달
4. 잠시 후, A이든 또는 B, C 이든 어느 프로세스가 똑같은 블록을 다시 요청 가능!
5. 서로 다른 프로세스들이 같은 영역을 요청 && 시간이 걸리는 디스크 액세스 작업 반복!

   ⇒ 이 때문에, 디스크 캐시 개념이 고안되었다.

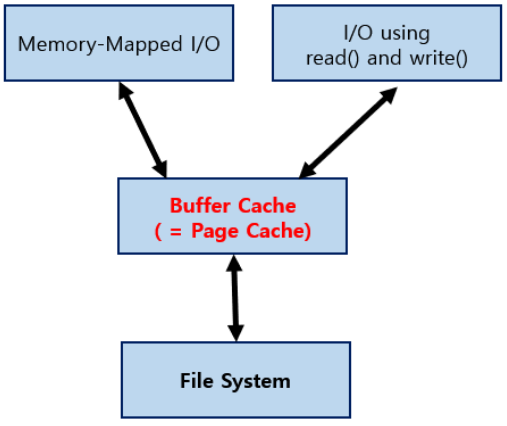
※ 블록은 논리적인 블록 디스크에서는 섹터를 의미

※ 섹터는 512byte, 최근에는 Buffer Cache와 Page Cache가 통합 → page단위(4KB)를 사용

※ Unified Buffer Cache



* Unified Buffer Cache를 이용하지 않는 File I/O



* Unified Buffer Cache를 이용한 File I/O

⇒ Unified Buffer Cache : 버퍼와 페이지 캐시를 통합한 것, 최근엔 Unified Buffer Cache를 많이 사용

⇒ 버퍼 캐시의 단위 : 페이지와 동일(4KB)한 단위의 블록을 사용, 페이지를 용도에 맞춰서 할당

※ Unified Buffer Cache를 이용하지 않는 File I/O

- 파일을 open한 후 read, write 시스템 콜
  - 운영체제가 해당하는 파일의 내용이 Buffer Cache에 있는 지 확인
  - Buffer Cache에 존재한다면 전달해주고 없으면 디스크에서 읽어서 전달
  - 사용자 프로그램 → 자신의 주소 영역중에 있는 Page에다가 Buffer Cache 내용 복사해 사용

- Memory Mapped I/O
  - 운영체제에게 'Memory Mapped I/O 쓰겠다'라는 mmap 시스템 콜을 호출
  - 자신의 주소공간 중 일부를 파일에다가 맵핑 (Disk → Buffer Cache 읽는 작업은 동일)
  - 읽은 내용을 Page Cache에다가 카피
    - Page에 파일에 Mapped된 내용이 들어감, 이후 메모리 접근방식 통해 File I/O 처리
    - 맵핑 이후는 운영체제의 간섭없이 내 메모리 영역에다가 데이터를 읽거나 쓰는 방식
  - 만약 매핑만 해두고 메모리로 안 읽어왔다면 Page Fault가 발생

⇒ Buffer Cache에 있던 없던, 일단 운영체제에게 요청을 해서 받아오는 형식

※ Unified Buffer Cache를 이용하는 File I/O

⇒ 운영체제가 공간을 따로 나누어놓지 않고 필요에 따라서 Page Cache에 공간을 할당해서 사용

⇒ Buffer Cache와 Page Cache를 sharing해 사용하는 방법

- 파일을 open한 후 read, write 시스템 콜을 한다.
  - 해당하는 내용이 Disk File System에 있던 Buffer Cache(=Page Cache)에 있던 상관 X
  - 곧바로 운영체제에게 CPU 제어권이 넘어간다.

- Memory Mapped I/O
  - 처음에 운영체제에게 자신의 주소 영역중 일부를 파일에 매핑하는 단계 수행
  - 사용자 프로그램 주소 영역에 Page Cache가 매핑
  - Buffer Cache가 별도로 존재하지 않고 Page Cache에다가 읽고 쓸 수 있다.

⇒ 일단 Page Cache에 있다면 운영체제에게 도움을 받지 않고 I/O가 가능하다는 점
