26. Flash-based SSDs

(summary 시험 치기 전에 다시 듣기)

▼ Solid-State Storage(SSD)

- 기계적으로 움직이는 부분이 없음 (seek, rotation time 등이 존재하지 x)
- memory와 비슷한 부분이 많지만 전원이 꺼져도 정보가 유지됨!

  → flash를 사용하기 때문!

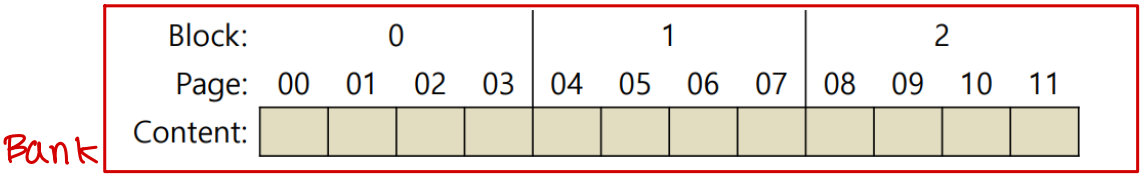
▼ Flash(NAND-based flash, 대부분의 SSD가 사용)

→ 여러 개의 transistor로 구성되어 있음

- single transistor : 하나 혹은 여러 개의 bit를 저장 가능
- transistor 내부에 저장할 수 있는 bit 개수에 따라서 종류가 나뉨
  - SLC, MLC, TLC
    single 2개 3개

▼ flash chips

→ 한 bank 안에 여러 개의 block 존재
(or planes)



- 각 bank → 굉장히 많은 수의 block 존재
  ↳page보다 크기가 상당히 큼
  - blocks : 128KB or 256KB
- 각 block → 굉장히 많은 수의 page 존재
  - pages : 4KB

▼ Basic Flash Operations

↳low level에서 발생하는 op

⇒ 속도 : Read > Program > Erase
cost : < <

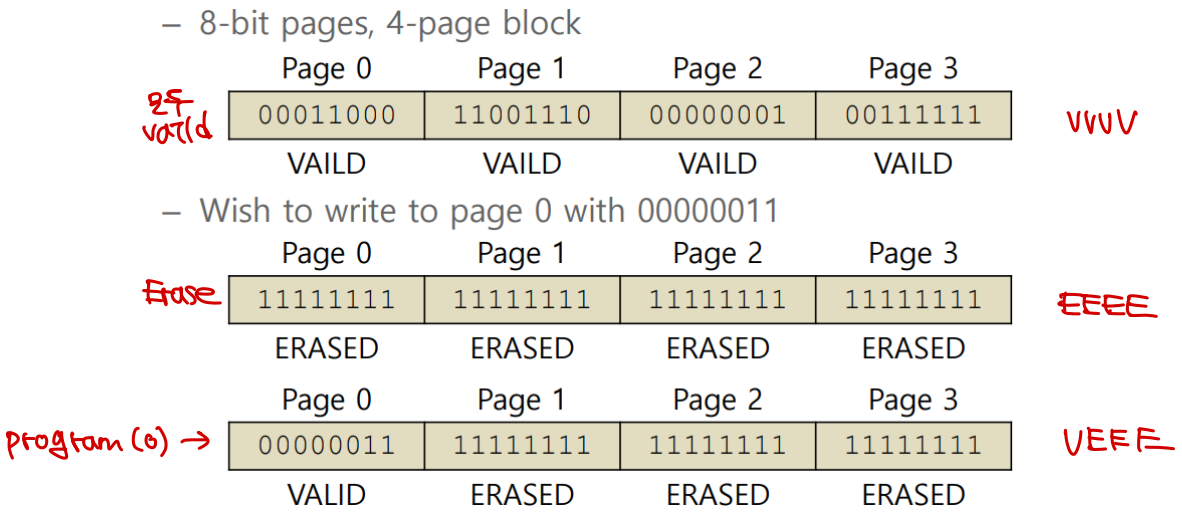
1. Read (2kB or 4kB)
    - page number로 어떤 page든 읽을 수 있음
    - device의 위치에 상관 없이 꽤 빠른 operations → 10ms
2. Erase → flash를 사용할 때 가장 큰 문제점
    - flash 내부의 page에 write하기 위해 먼저 block 전체를 erase 해야 함
        - 모든 block의 value bit를 1로 바꿈 → few mileseconds!!
    - 가장 비싼 operations ☆
3. Program
    - write와 비슷함
    - 일단 block이 erase되면 program command는 0인 page들을 1로 바꾸는데 사용할 수 있음
    - 100ms 정도 걸리는 operations

- a simple example

		iiii	Initial: pages in block are invalid (i)
Erase()	→	EEEE	State of pages in block set to erased (E)
Program(0)	→	VEEE	Program page 0 state set to valid (V)
Program(0)	→	error	Cannot re-program page after programming
Program(1)	→	VVEE	Program page 1
Erase()	→	EEEE	Contents erased; all pages programmable

block 단위로 write
↳ 이전의 page
→ 이미 Erase 했어야 오류남

- a detailed example



→ block 내에 어떤 page에 overwriting하기 전에 무조건 어떤 data든 다른 위치로 옮겨주고 시작해야 함

(page는 overwrite 될 수 없음)

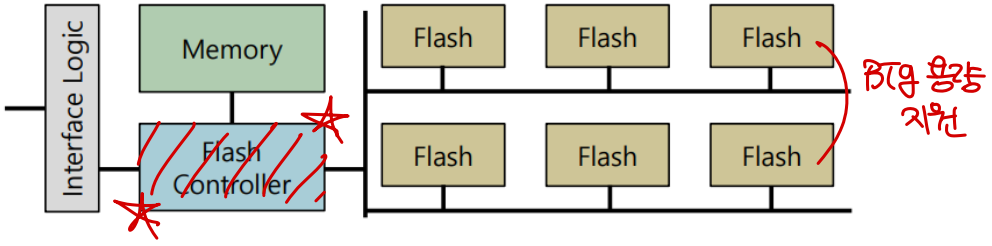
▼ block 기반의 storage device에서 flash chip인 SSD는 어떻게 수행?

- SSD
  - flash chip의 집합
  - volatile memory(SRAM) → caching과 buffering에 유용
  - control logic

    ⇒ FTL 사용!

▼ FTL(flash translation layer)

→ SSD 내의 CPU라고 생각하자



- logical block에 대한 r/w 요청 받음

  → low-level의 read, erase, programm command로 바꿈

*Bad approach

- physical page에 대한 mapping은 어떻게?
  - ① N개 logical page read
    - physcial page N개의 direct하게 mapping
  - ② N개 logical page write
    - page N이 속한 전체 block read
    - erase the block
    - 이전 page, 새 page 모두 program(write)

  → overhead가 높음

- problem
  - performance problem
    - write amplification : 한 byte 쓰기를 하더라도 한 page (4KB) 쓰기 해야함
  - reliability problem : 같은 block에 대해서 계속 erase하면 닳아버림
    - wear out → 사용 가능한 용량이 줄어듦

문제점 어떻게 해결 ??

▼ A Log-Structured FTL

FTL 문제점을 해결해보자!

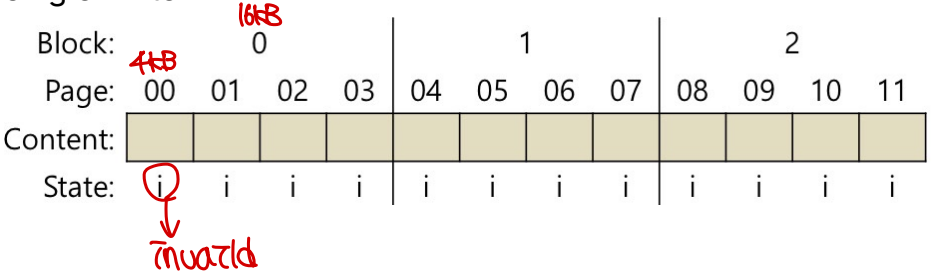
⇒ write하려는 것들을 모아서 한꺼번에 새로운 공간에 write

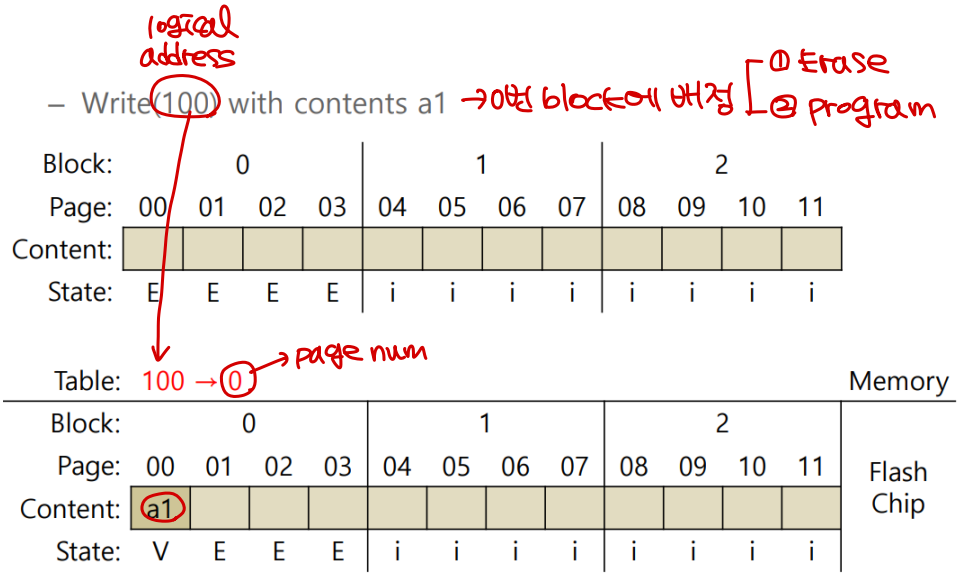
- 현재 write의 대상이 된 block : 다음 사용 가능한 위치에 write 추가

⇒ 계속 위치가 바뀜

- mapping table : system의 각 local block의 물리 주소를 저장

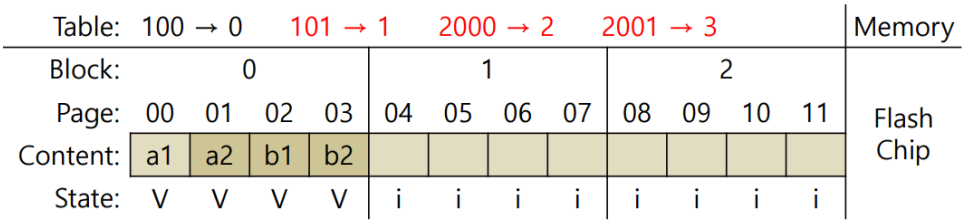
- example
  - assumptions
    1. client : 4KB 사이즈 만큼 read or write
    2. SSD : 4개의 4KB page ⇒ 16KB size의 block을 포함함
  - single write





- multi write

  - Write(101) with contents a2
  - Write(2000) with contents b1
  - Write(2001) with contents b2



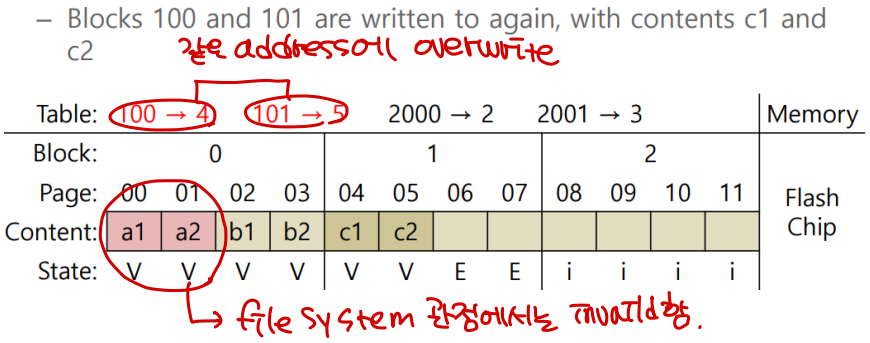
advantages

1. erase 덜 자주 발생 ⇒ 성능 향상
2. wear out 되는 정도를 균등하게 맞춰줌 → device의 lifetime 증가 ⇒ wear leveling

disadvantages

1. garbage collections → 반드시 필요함. 이전의 data를 회수하는 것이 필요
   a. overwrite 하게 되면 garbage 생김
2. high cost of in-memory mapping tables : 전원이 꺼지면 mapping 정보 사라짐
   a. mapping table의 persistency 조절 (Out-of-Band, OOB)
   b. device가 클수록 table이 더 많은 memory를 필요로 함

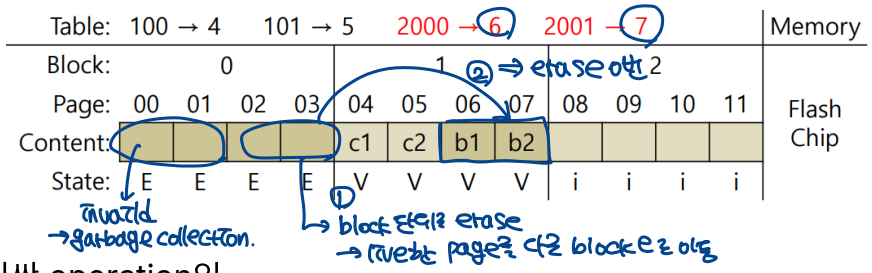
- Garbage Collection
  - example



  - basic process

→ mapping page를 통해 찾음

1. 하나 혹은 그 이상의 garbage page를 가지고 있는 block 찾음
2. 그 block 내의 live한 page 읽음 → garbage가 아님
3. live한 page를 log에 write ⇒ 다른 block에 작성 (+ mapping table update) 이후 Erase
4. block 전체를 가져와서 writing에 사용 ⇒ erase한 뒤!



⇒ 비싼 operation임

→ system이 idle할 때 실행 혹은 garbage가 많이 모인 table collection 하도록!

- Mapping table size
  - 1-TB SSD
    - 4B entry per 4KB page ⇒ memory mapping을 위해 1GB 필요

⇒ page level FTL은 효율적이지 않다!

entry 개수를 줄일 수 있음.

  - ★Block based mapping★

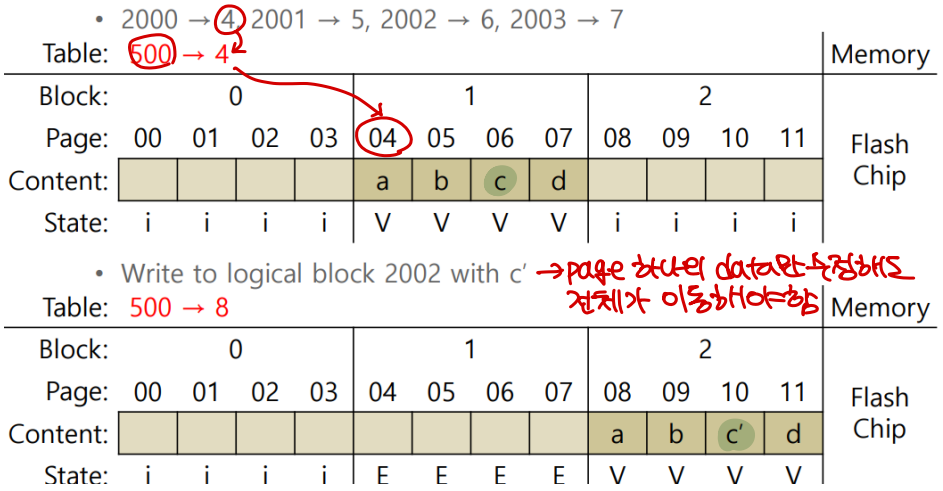
- page 단위 대신 block 단위의 pointer로 mapping
- logical block address : chunk number + offset
  - logical block 2000 : 0111 1101 0000
  - logical block 2001 : 0111 1101 0001

page offset

⇒ 가장 큰 문제 : page 하나에도 block 단위로 수정해야 함

→ 비효율적임

- example



page 하나의 data만 수정하더라도 전체가 이동해야함

- Hybrid mapping ⇒ page + block
  - Log table: a small set of per-page mappings
  - Data table: a larger set of per-block mappings

  ⇒ advantage
  - 특정 logical block 찾을 때 log table (①) → data table (②) 순으로 찾음
  - FTL : log table에서 data table로 옮길 수 있는 애들이 없는지 수시로 확인 + 이동

- page mapping + caching
  - cache : active한 부분에만 적용
  - 작은 page 모음에만 접근하도록 주어진 workload에서 잘 작동함

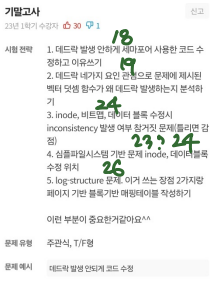
↳ flash에서 발생하는문제

▼ Wear Leveling

muitilple erase → flash block이 wear out

⇒ FTL : 이 부분 개선하도록 노력해야 함

- 모든 block은 동시에 wear out될 수 있음
- basic log srucuring approach
  - write 작업을 분산하는데 좋은 초기 작업
  - garbage collection을 위해 data를 자주 새로운 곳에 쓰려는 경향이 있음
    - 계속 같은 곳에만 사용하기 때문에 erase 되는 아이들이 적음



- 만약 계속 overwritten되지 않는 오래 사는 data들로 block이 가득 차있다면?
  - garbage collection이 절대 block을 reclaim하지 않음

→ write 작업이 불균등함

⇒ FTL : 이러한 block들로부터 live한 data 주기적으로 읽은 뒤 다른 곳에서 다시 write 해야 함

*기말예상

1) semaphore deadlock 발생?

↳ circular dependency ⇒ 서로의 자원을 서로에게 기다림. ⇒

- Mutual exclusion ⇒ lock free
- Hold-and-wait : lock 잡고있는데 또 잡으려고 함. ⇒ lock 한 번에 얻음
- No preemption : lock을 강제로 뺏어올 수 X ⇒ trylock 사용.
- Circular wait ⇒ lock 얻는 순서 정의

2) inconsistency 발생여부

inode, bitmap 성공여부 따라지면 ∴ inconsistency 발생

3) simple file system

4) log-structure FTL.

① erase가 적어져 performance 향상

② wear leveling

LFS FTL ☆.