25. Log-structured File System

▼ 등장 배경

1. system memory → 계속 커짐
   a. (page) cache memory도 커짐 ⇒ 더 많은 data가 cached → 매번 disk 접근 X
   b. disk traffic에 의해 write 성능은 read만큼 좋아지지 않음
      ↳ load가 필연적이기에 disk 접근이 많
2. random I/O와 sequential I/O 사이에 너무 큰 gap

   → seek과 rotational delay costs는 여전히 너무 성능 bad임

   ⇒ random I/O 성능이 bad
3. 현존하는 file system들이 대부분의 workload에서 성능 거지 똥임

   ex) file system : one block의 새로운 file을 생성할 때 많은 수의 wirte를 수행
   ↳ 고작 file을 하나 생성하면 성능 증가 X

▼ Log-structured File system(LFS)

disk에 writing 할 때 in-memory segment에 쓰고자 하는 data 정보 모두 모아 놓음

→ segment가 꽉 차면 disk에서 sequential하게 붙어 있는 sector에 한 번에 작성

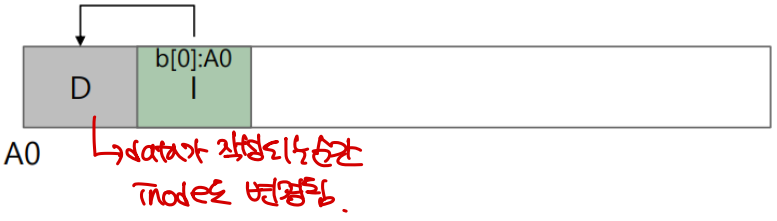
(기존에 있는 data overwrite x, 새로운 공간에 계속 작성)
↳ free list

- 최근 연구 : large I/O → 높은 성능의 flash-based SSD들을 필요로 한다는 것을 보임
  ↳ 다음 chapter

▼ Writing to disk

- file-system에 어떻게 sequential하게 write를 한 번에 모아서 함?
  - data만 쓰게 되면 inode는 random하게 여기 저기 작성될 수 있음 (결국 random I/O)

    ⇒ metadata(inode)도 함께 작성!☆ → inode 위치가 계속해서 변경됨 (고정된 위치에 저장 X)



↳ data가 갱신되면 순간 inode도 변경됨.

- 단순하게 sequential writing만 하는 것은 성능 개선에 큰 도움이 되지 X

  Time T : A2는 처음에 하나의 block을 써야 하는데 걸리는 시간

  Time T+ε : A1 "

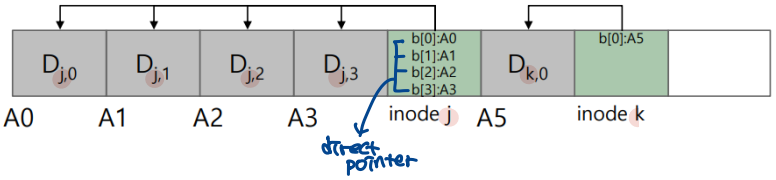
  Disk ⇒ ($T_{rotation}$ - ε) 만큼 기다려야 sector에 data 작성 가능

⇒ but, 큰 사이즈의 write를 한다면 성능적인 이득이 있을 것임

- write buffering
  - LFS는 되게 큰 memory 영역에다가 request를 차곡차곡 모음
  - 충분할만큼 모였을 때 한 번에 disk에 작성

⇒ segment : request가 모인 memory 영역 (크기는 few MB)

⇒ 디스크에 한 번에 request 전송



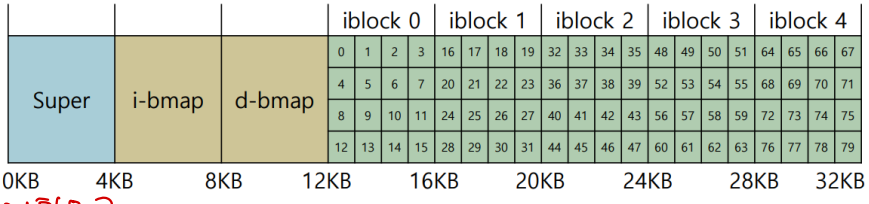
→ segment size는 환경에 따라 다른 값으로 설정됨

▼ Finding inodes → Inode Map

LFS : 아예 새로운 block에 작성 → sequential한 I/O 작성하기 위함

→ 특정 inode number를 갖는 inode 위치가 계속 바뀜 (디스크 전체에 흩어져 있음)

⇒ update 꾸준히 해줘야 함

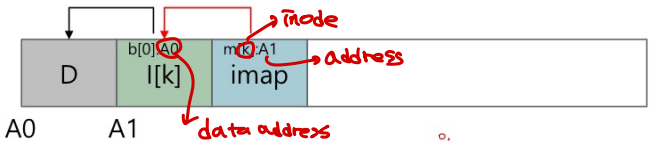


* inode 어케 찾음?

- **imap(Inode Map)**
  - inode number를 받아서 가장 최신 version의 inode로 disk address에 기록해놓는 table
    - inode가 disk에 쓰일 때마다 imap update되어야 함

      새로운 위치로!

    → 고정되어 있지 않음!
  - imap은 disk 어디에 존재?

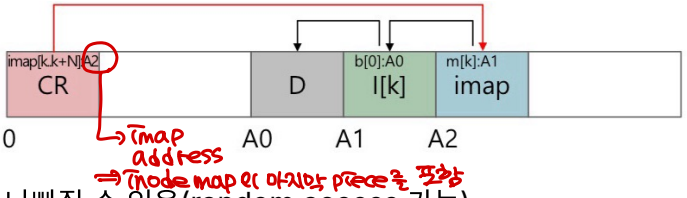


- disk 내에서 고정해놓는다면? → performance 나쁨(여전히 random I/O)

⇒ update된 내용만 imap에 저장해놓기

- checkpoint Region(CR)
  - imap도 고정해놓는다면? → 여전히 random access

    ⇒ checkpoint region 사용!
  - CR ⇒ 고정된 위치에서 사용



    - 부분적으로 저장해놓음
    - CR을 매번 update하면 성능이 나빠질 수 있음(random access 가능)

      ⇒ periodically하게 update(ex. 30초에 한 번)

      but, 이 사이에 crash 발생하면 update 중단되고 정보 손실 가능

▼ Reading a file from disk (w. imap)

1. checkpoint region 읽기
2. CR 이용해서 전체 inode map 읽은 뒤 memory에 넣어둠

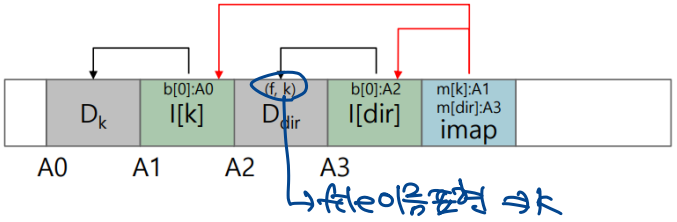
⇒ 매번 disk 접근하지 않기에 추가적인 overhead는 미미함

▼ Directory 접근

기존 system은 작은 file을 계속해서 생성할 때 좋은 성능을 보여주지 않음

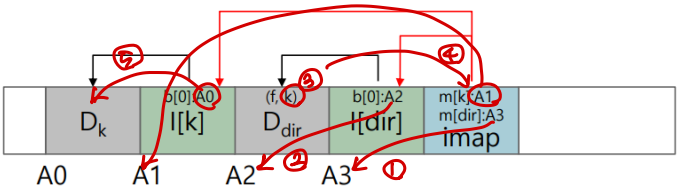
→ LFS ver 확인해보자!

1. Creating a file dir/f ⇒ directory 고려해서 file 생성된다고 생각해보자



- LFS : direcotry에 해당하는 inode update
  - director data 영역에 file 영역 이름 표현해야 함

2. accessing file f



1. memory에 저장되었던 inode map을 통해 dir의 inode 위치 찾음 → A3
2. dir inode 읽은 뒤 dir data의 위치 찾음 → b[0]: A2
3. data block 읽은 뒤 (f,k)에 매핑되어 있는 inode-number의 name 찾음 → (f, e) → file name
4. inode number k에 위치한 inode map 찾음 → m[k]: A1
5. 그 address에 있는 data block read → b[0]: A0

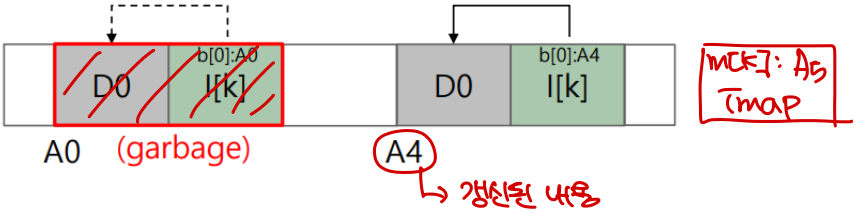
▼ Garbage Collection →

LFS : 매번 data를 새로운 위치에 씀
⇒기존에 존재하던 data?

회수가 되지 않은 영역을 모아서 중간에 update 해줌

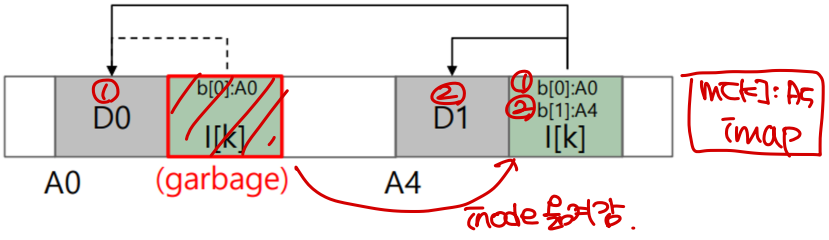
→ data, inode, 다른 자료구조들의 오래된 version 찾은 뒤 초기화

- example
  - update data block



  - appending a block to file k

↳data 추가



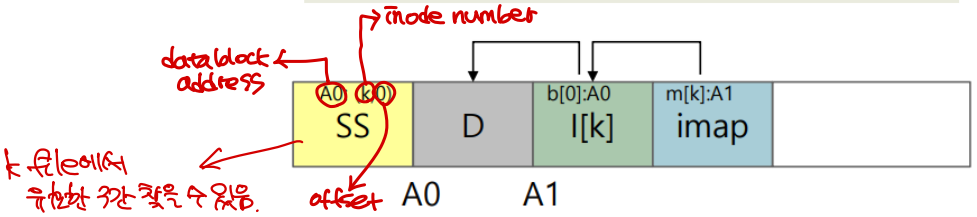
⇒ 많은 양의 공간을 clean up하게 되면 연속적인 writing 가능

① 기존의 old segment clear
② 어떤 block이 사용되는지… live한 block 옮겨감
③ 기존의 새로운 segment로 compact → 새로운 데이터 disk에 기록 (N<M)
④ 기존의 clean segment free → 연속적인 write를 위한 공간으로 사용

- 살아있는 block인지, 죽어있는 block인지 어떻게 check?

  ⇒ segment summary block

```
(N, T) = SegmentSummary[A];
inode = Read(imap[N]);
if (inode[T] == A)
    // block D is alive
else
    // block D is garbage
```



- 어떤 block을 초기화 + 언제 초기화?
  - 언제 초기화?
    - idle time ⇒ cpu가 여유있을 때
    - disk가 꽉 찼을 때
  - 어떤 block을 clean?
    - hot segments : 자주 overwritten
    - cold segments : 몇 가지 dead block을 가지고 있지만 아직 살아있는 놈 몇 개 있는 segment

  ⇒ cold 초기화 한 뒤 hot 초기화

▼ Crash Recovery

LFS가 disk에 쓰고 있는 동안 crash 일어난다면? → journaling 호출

1. CR에 쓰고 있을 때 발생

   ⇒ LFS는 두 가지 CR을 가지고 있으며 번갈아 사용함

   a. header에 write (with timestamp)

   b. CR의 body에 작성

   c. last block에 write(with timestamp)

   → 항상 일관적인 timestamp를 가진 가장 최근의 CR을 골라서 사용함

2. segment에 쓰고 있을 때 발생

   ⇒ LFS는 30초마다 update 하기에 file system이 이 중간에 침입 가능

   → roll forward