24. FSCK and Journaling

bitmap, *inode* ... 등 write 작업이 발생했을 때

→ 실제로 disk까지 wirte operation이 완벽히 종료되지 않을 수 있음 (crash가 일어날수 있음)

⇒ 어떻게 해결?

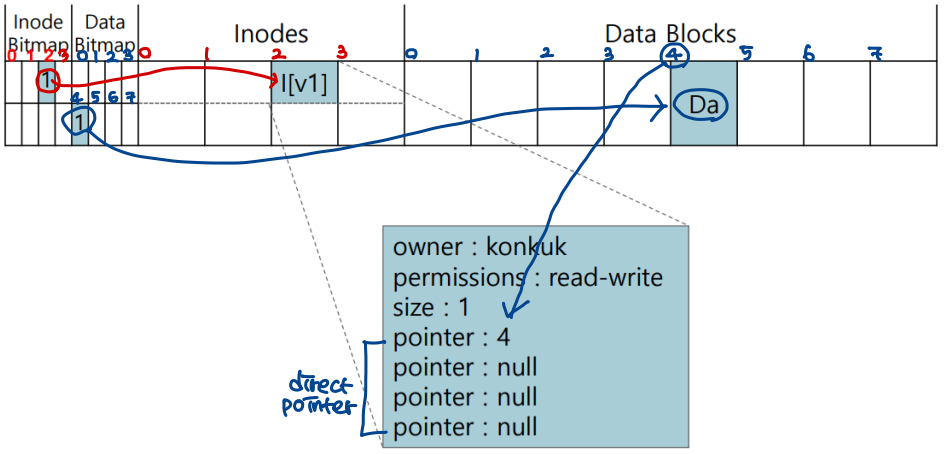
▼ How to Update the Disk despite Crashes

- system → write 도중에 crash 혹은 lose power 일어날 수 있음
  - on-disk state에서 발생한다면 부분적으로만 update 될 것임
  - crash 이후에는 system이 file system을 다시 mount 하고 booting시키려 할 것임.

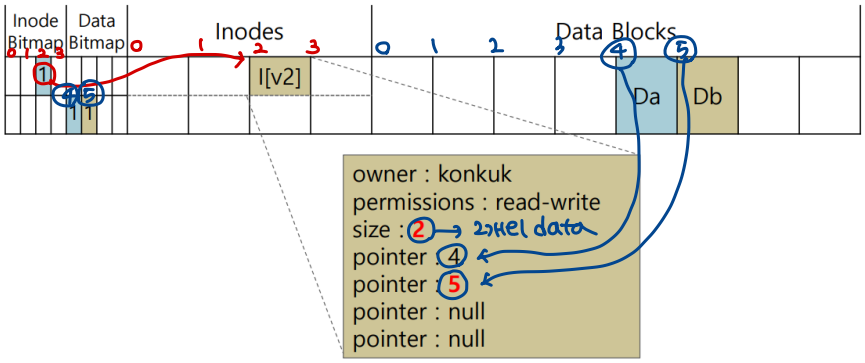
⇒ 어떻게 상태 유지? → ① file system checker, ② journaling

▼ example of crash

① write operation



- 이미 존재하는 file에 하나의 data block 추가할 때
  1. open the file
  2. lseek() → file의 끝으로 offset 옮김 ⇒ 끝에 위치하는 (가장최근에 추가한) data block을 inode에 표기
  3. close하기 전에 file에 4KB write



- Crash scenarios
  - disk write 종류
    1. data bitmap
    2. *inode*
    3. data block

1. single write만 성공했을 때
   a. data block만 wirte 성공
      → consistency 문제는 발생하지 않음, but 존재하지 않는 것으로 인식
   b. *inode*만 write 성공
      → disk로부터 garbage data 읽게 됨
      → inconsistency 발생
   c. bitmap만 write 성공
      → inconsistency 발생 + space leak 발생
2. two write 성공, 하나는 실패했을 때
   a. data block만 write 실패
      → data block에 garbage data 저장
   b. bitmap만 write 실패
      → inconsistency 발생(overwrite 될 수 있음)
   c. *inode*만 write 실패
      → inconsistency 발생

▼ FSCK(file system checker)

a UNIX tool for finding file-system inconsistencies and reparing them

⇒ inconsistency가 일어나도록 냅둔 다음에 나중에 고침(리부팅할 때)

- 모든 문제를 해결하는 것은 x
- 목표 → file system metadata가 내부적으로 consistent하게 만들기!

▼ fsck가 하는 일

1. superblock check
2. free block check
   a. inode, indirect block, double indirect block scan check
      → file system 내부에 현재 존재하는 block들 check하기 위해!
   b. 만약 bitmap, inode 사이에 inconsistency가 발견된다면
      → inode 내부의 information 바꿔서 해결 ex) inode free 혹은 bitmap 고쳐넣기 등...
3. *inode* state
   a. 각각 할당된 inode가 모두 vaild하도록 만들어줌
      ex) regular file, dir, symbolic link
   b. 만약 쉽게 고쳐지지 않는 inode field가 있다면 초기화
      → bitmap도 함께 수정해줌
4. *inode* links
   a. 할당된 *inode*의 *link* count verify (reference count)
5. duplicates
   a. 같은 block에 대한 두 개의 다른 inode가 있을 경우 check
      → bad inode를 ① clear 혹은 ② block copy
6. bad block pointers
   a. vaile한 range 내에서 무언가 문제가 생겨서 'bad'한 pointer로 여겨짐
   → pointer 초기화
7. directory checks
   a. 각 dir의 content check → "." or ".." 이 첫번째에 entry에 있어야 함.
      ⇒ 각 entry를 가리키는 inode가 할당되어 있음.

▼ Drawbacks of fsck

too slow : 너무 큰 disk라면 scanning하는게 몇시간씩 걸림

⇒ wasteful : 문제를 해결하기 위한 cost가 너무 비쌈

▼ Journaling(write-ahead logging)

- journaling
  - disk를 update할 때 어떤 내용을 update할 건지 note를 남겨 놓는 것
- checkpointing → 실제로 적용하는 단계
  - metadata와 data를 진짜 file system에 update하는 순간
    (fsck는 지금까지 이걸 해온 것임)

▼ Ext3

block group이 여러 개 존재할 수 있는 disk 구조

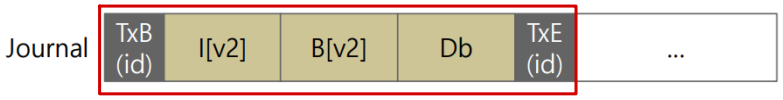
↔ 지금까지 본 것은 하나만 존재!

→ 각 block group : inode bitmap, data bitmap, inodes, data blocks 포함

⇒ journal이 존재 : data update할 때 내용을 남겨 놓음

▼ Data journaling

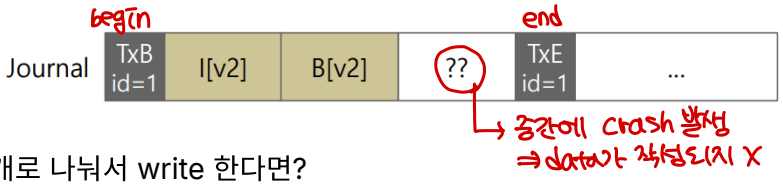
* transaction : Journal에 내용을 update



- writing the journal
  - 매번 write하고 매번 끝날 때까지 기다린다면?

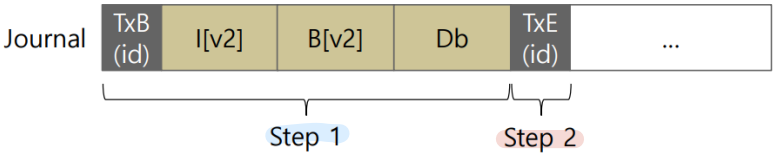
    → 느림
  - 다섯 개의 block을 한 번에 write한다면?

    1. reordering(disk scheduling 다시 해야 함)
    2. journal에 write 하는 중간에 crash일어날 수 있음



  - step을 두 개로 나눠서 write 한다면?

    ↳ overhead는 발생하지만 완전함



    - step1 : TxE block을 제외한 모든 block에 write (디스크에서 작성순서 결정)
    - step2 : step1이 끝나면 TxE block에 write

⇒ TxE 에 작성할 때에는 atomic하게 실행되어야 함
→ 부분적으로 작성되는 것을 막기위해 TxE는 한 섹터의 크기를 가짐(512B)

▼ Recovery → 복구는 어떻게?

1. crash가 transaction 이전에 발생(= journal이 제대로 쓰여지지 않았다면)

   → 그냥 update skip

2. crash가 transaction 이후에 발생(= checkpoint가 끝나기 전)

   a. log을 scan한 뒤 disk에 commit된 transaction 찾음

   b. journal 안에 있는 정보를 반영하여 원래 쓰려던 곳에 마저 write 시도함 → transaction replay

      i. 하지만 recovery하다가 불필요하게 반복적으로 update 할 수 있음

▼ Batching log updates

- problem : 많은 양의 disk traffic 발생 가능

  ex) creating two files in the same dir

  ⇒ 이 file들에 대한 inode가 같은 inode block이라면 같은 block에 계속 쓰고 또 씀

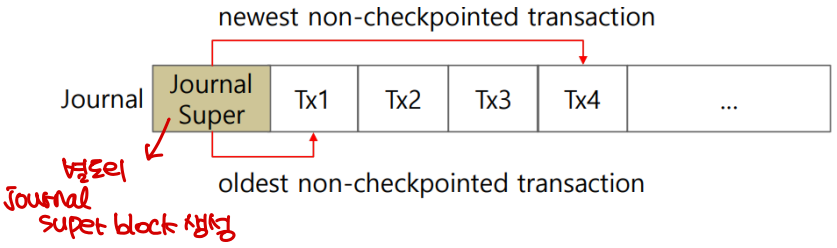
- solution : buffering update → 묶어서 한 번에 요청
  - file system buf : 중간 중간에 memory에 update

    → disk traffic 막을 수 있음

▼ Making the Log Finite

- problem : log가 꽉 찰 수 있음
  - log가 클수록 recovery가 오래 걸림
  - 반대로 너무 log가 작다면 많은 것을 쓰지 못 함

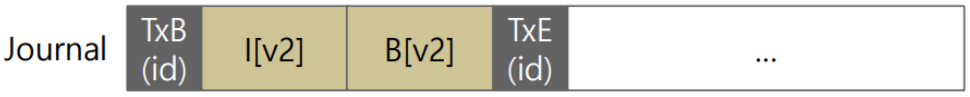
- solution : circular log → circular queue와 동일한 구조
  - 일단 transaction이 한 번 checkpoint되고 나면 file system이 공간을 free함



▼ Ordered Journaling(= metadata journaling)

- problem : data journaling +) data를 어디에 작성?...
  - disk에 write할 때마다 journal에 먼저 쓰기 때문에 double write traffic 발생

- solution : metadata journaling
  - user data → journal에 쓰이지 않음



  - crash할 때에는 어떻게?
    - 몇몇 file system : 관련된 metadata를 disk에 쓰기 전에 disk에 먼저 data block write ⇒ journaling 하기 전에 data 먼저 써버림

- basic protocol
  1. Data write : 실제 data write
  2. Journal metadata write (step1 →)
  3. Journal commit (step2 →)
  4. Checkpoint metadata
  5. Free ← crash가 일어나지 않았다면 superblock 정보 update free