Lecture Note #12 - Chapter 41. Locality & FFS

(1) Background

- Unix file system

- Super block (S) : 전체 파일 시스템에 대한 정보

- 전체 볼륨의 크기
- i-nodes 가 얼마나 있는지?
- free list of blocks의 헤드를 가르키는 포인터

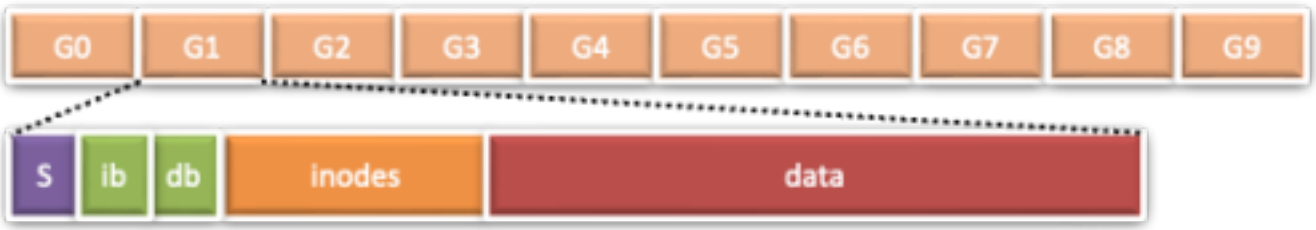
- UFS의 단점
  - ✓ 성능이 최악
  - ✓ 데이타 블럭이 랜덤하게 할당됨
    - · Fragmentation
  - ✓ 디스크로부터 데이타 전송이 비효율적
  - ✓ 블럭 크기가 너무 작음 (512 bytes)
  - ✓ i-nodes 가 데이타 블럭과 많이 떨어진 곳에 할당
    - · 모든 i-node 데이타와 많이 떨어진, 디스크 시작 부분에 위치

(2) FFS

- Design the file system to be 'disk-aware' 디스크 친화적인 . 디스크의 구조를 생각…. => 동일한 인터페이스를 유지하면서 내부 구현을 변경

- 기본 아이디어

- 관련 inode 및 데이터 블록을 함께 배치 : 긴 seek time 방지
- 디스크를 실린더 / 블록 그룹으로 나눔



- 파일 액세스 패턴 관찰 결과 동일한 디렉토리에 있는 파일에 액세스 할 확률 : 40 % => Name-based Locality
  - ✓ 이름 기반 구역성……절대 경로
  - ✓ 가능 한 경우 모든 파일을 단일 그룹의 디렉토리에 배치
    - · 예 : / foo / bar, / foo / boo, / foo / baz를 같은 그룹에 배치
  - ✓ 그룹간에 디렉터리 수 균형 조정
    - · 예 : 디렉토리 수가 적거나 사용 가능한 inode 수가 많은 그룹에 새 디렉토리를 배치

ext2

Lecture Note #12 - Chapter 41. Locality & FFS

지역성과 Fast File System

(1) Background

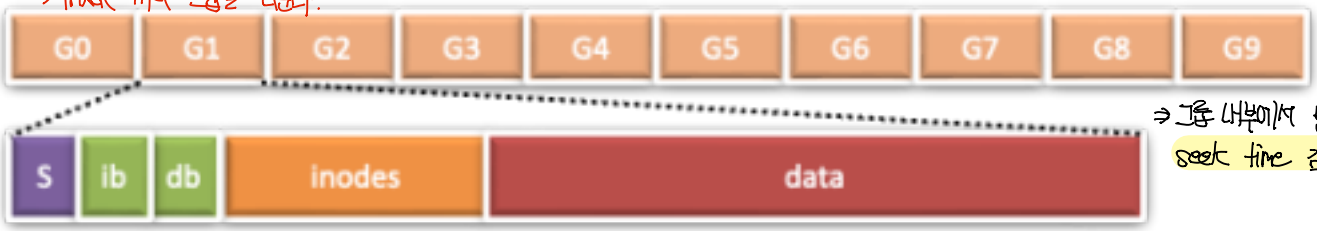
- Unix file system (UFS)

- Super block (S) : 전체 파일 시스템에 대한 정보
  - 전체 볼륨의 크기 (UFS의 크기)
  - i-nodes 가 얼마나 있는지? (inode의 개수)
  - free list of blocks의 헤드를 가르키는 포인터 (비어있는 블록을 가리키는 …)
- UFS의 단점
  - ✓ 성능이 최악
  - ✓ 데이터 블럭이 랜덤하게 할당됨 , 데이터 블록을 찾는 시간 "seek time"이 늘어나서 비효율적
    - Fragmentation (파일의 크기도 점점 커지기도 해서)
  - ✓ 디스크로부터 데이타 전송이 비효율적
  - ✓ 블럭 크기가 너무 작음 (512 bytes)
  - ✓ i-nodes 가 데이타 블럭과 많이 떨어진 곳에 할당
    - 모든 i-node 데이타와 많이 떨어진, 디스크 시작 부분에 위치

요즘 device는 BSD 기반이라 FFS가 사용됨
⇒ 더 많은 용량은 Android와 필요로 하는 애가 바로 이것

(2) FFS ⇒ multimedia 파일에 아주 좋다 (파일 전체가 메모리 크기가 매우 유리함)

- Design the file system to be 'disk-aware' 디스크 친화적인 . 디스크의 구조를 생각…. => 동일한 인터페이스를 유지하면서 내부 구현을 변경
- 기본 아이디어 - Seek time을 어떻게 줄일까?
  - ① Block의 크기를 늘린다. UFS 512B → FFS 1KB / UFS 4KB → FFS 16KB
    - ↳ 단점: Internal fragmentation이 생기면?
      ⇒ 시간이 지나면서 디스크 여유가 떨어짐
      ⇒ 그럼 Internal fragmentation은 고려 X
  - ② - 관련 inode 및 데이터 블록을 함께 배치 : 긴 seek time 방지
  - - 디스크를 실린더 / 블록 그룹으로 나눔 (그룹화)
    - → track 끼리 그룹을 나눈다.

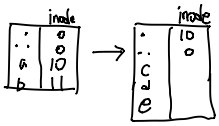


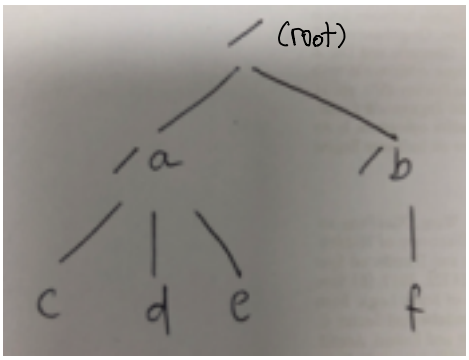
⇒ 그룹 내부에서 탐색하므로 seek time 감소

- 파일 액세스 패턴 관찰 결과 동일한 디렉토리에 있는 파일에 액세스 할 확률 : 40 % => Name-based Locality (이름 기반 근접성)
  - ✓ 이름 기반 구역성……절대 경로
  - ✓ 가능 한 경우 모든 파일을 단일 그룹의 디렉토리에 배치 ⇒ 같은 디렉토리에 있는 파일은 모두 같은 그룹에 넣어야 seek time ↓
    - 예 : / foo / bar, / foo / boo, / foo / baz를 같은 그룹에 배치
  - ✓ 그룹간에 디렉터리 수 균형 조정
    - 예 : 디렉토리 수가 적거나 사용 가능한 inode 수가 많은 그룹에 새 디렉토리를 배치

(3) Allocate Files and Directories in FFS

- /, /a, /b directories
- c, d, e, f : normal files

/a/e ⇒ 



- 그룹은 0~7까지 8개 존재

```
group inodes      data
    0 /---------- /----------
    1 acde------- accddee---
    2 bf--------- bff--------
    3 ----------- -----------
    4 ----------- -----------
    5 ----------- -----------
    6 ----------- -----------
    7 ----------- -----------
```

) 디렉토리 별 그룹

✓ 모든 그룹에 공평하게 배치 -> 성능 떨어짐

```
group inodes      data
    0 /---------- /----------
    1 a---------- a----------
    2 b---------- b----------
    3 c---------- cc---------
    4 d---------- dd---------
    5 e---------- ee---------
    6 f---------- ff---------
    7 ----------- -----------
```

✓ 하나의 하나의 그룹에 큰 파일을 다 넣을 경우 - 다른 파일은 동일한 그룹에 할당 불가 분산의 효과가 떨어짐

```
group inodes      data
    0 /a--------- /aaaaaaaaa aaaaaaaaaa aaaaaaaaaa a----------
    1 ----------- ---------- ---------- ---------- ----------
    2 ----------- ---------- ---------- ---------- ----------
```

✓ 대용량 파일은 몇개의 그룹에 chunk 단위로 분산 배치

```
group inodes      data
    0 /a--------- /aaaaa----- ---------- ---------- ----------
    1 ----------- aaaaa------ ---------- ---------- ----------
    2 ----------- aaaaa------ ---------- ---------- ----------
    3 ----------- aaaaa------ ---------- ---------- ----------
    4 ----------- aaaaa------ ---------- ---------- ----------
    5 ----------- aaaaa------ ---------- ---------- ----------
    6 ----------- ---------- ---------- ---------- ----------
```

· 연속되게 못 읽음 -> 성능 저하 -> Chunk의 크기 조절 (충분히 크게하여 Chunk로 이동하는 시간 비율을 낮게 유지) =>

알아서 적절하게 조절해야 한다.

Amortization (양도) - 한쪽에 데이터가 너무 많으면 다른 Chuck로 넘김
- 이 또한 알아서 분배해야 한다.

- 기타 성능 향상

- Parameterized placement (파라미터 배치) : 디스크 레이아웃 최적화. 예전에는 순차 읽기로 인해 각 블록에 대해 전체 회전이 발생. 예 : 블록 1을 읽은 직후 헤드는 이미 블록 2의 시작 부분 지남

(FFS Standard vs. Parameterized Placement)

- Sub-blocks : 512-byte 작은 블록을 만들어 파일에 할당 --〉. 4KB·····4KB 16KB 〉 내부 단편화 작게
- 긴 파일 이름 〉 8 chars
- 원자적 rename()
  - ✓ 3번의 시스템 콜 -〉 임시 이름으로 이전 파일 둠
  - ✓ 1번의 시스템 콜

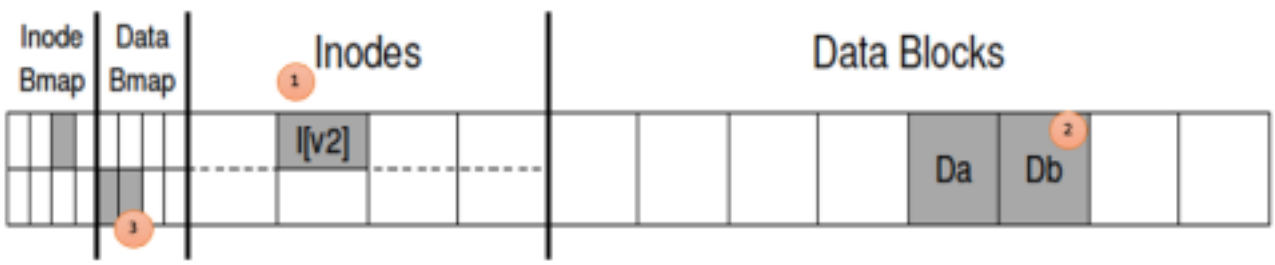
Chapter 42. Crash Consistency : FSCK & Journaling

File System Check

(1) Background

- 전원 결함이나 시스템 결함에도 불구하고 파일 시스템 데이터 구조는 지속성이 있어야 함
  - 디스크의 내용을 안전하게 갱신하는 문제
  - Crash Consistency (깨짐 일관성) 문제
  - FSCK 나 Journaling(write-ahead logging) 기법 등을 사용하여 Crash Consistency 해결

- Detailed Example
  - 기존 파일에 block 하나를 append하는 경우 - 갱신 도중에 crash가 일어나면?



(2) Crash Scenarios

① • data block (Db)만 디스크에 썼을 때 → inode와 bitmap 갱신을 못한 경우 → 다날라감 → Consistent
② • 갱신된 inode (I[v2]) 만 디스크에 썼을 때 → Inconsistent
③ • 갱신된 bitmap 만 디스크에 썼을 때
④ • inode (I[v2]) 와 bitmap 을 썼을 때
⑤ • inode (I[v2]) 와 data block (Db) 를 썼을 때
⑥ • bitmap 과 data block (Db) 을 썼을 때

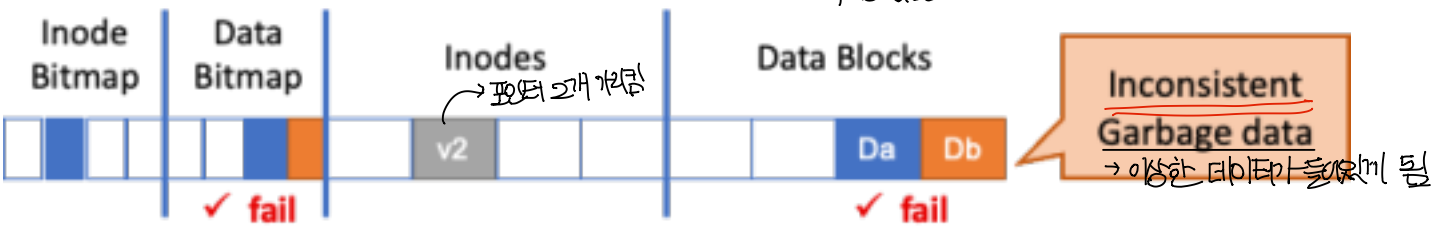
- File System의 Inconsistency
  - 파일 시스템에 대한 갱신은 원자적으로 수행
- Inconsistent State
  - ✓ inode가 비트 맵에 의해 할당되지 않은 데이터 블록을 참조
  - ✓ 사용되지 않은 데이터 블럭을 위해 비트맵 할당
- inconsistent file system의 결과
  - ✓ 잠재적인 Space leak
  - ✓ Inode가 가비지 데이타를 가르킴
  - ✓ 고아 데이타가 디스크 상에 남아 있게 됨

★ FS Ck는 Inconsistent한 상태만 복구 가능

bitmap과 Inode의 정보가 둘다 갱신되지 않았거나 둘다 갱신되었다면 ⇒ Consistent!!

file size 4KB

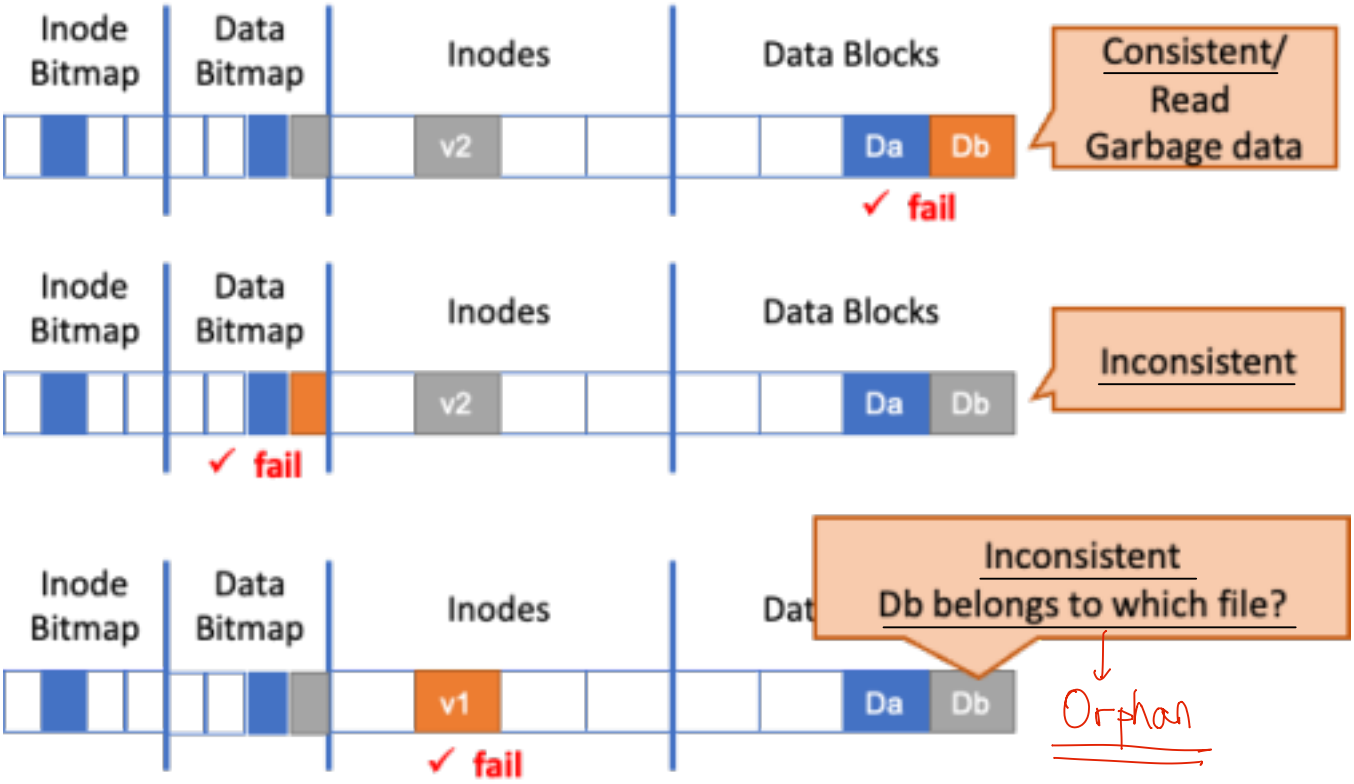
file size 8KB



→ 포인터 고개 갸우뚱

→ 이상한 데이터가 들어가게 됨

→ 공간 낭비
→ bitmap이 사용중이 있으므로 아무도 Db 사용 못하게됨



(3) The Crash Consistency Problem

- 파일시스템의 갱신은 원자적으로 이루어져야 함

- 일관성이 없는 상태 (Inconsistent States)
  - ✓ Inode가 bitmap에 의해 할당되지 않은 데이터 블록 참조 (Inodes xor Bitmap) = True 이면 Inconsistent
  - ✓ 파일에 의해 사용되지 않는 데이타 블록에 비트맵 할당

- 일관성 없는 파일시스템으로 나타나는 결과

① - 잠재적 공간 누수 Potential space leak : bitmap OK, inode X, data block X → bitmap이 가리키고는 있는 사용 중가

② - 가비지 데이타를 가르키는 inode가 생김 : bitmap X, inode OK, data block X → inode가 이상한 데이터 가리킴

③ - 고아 데이타 : 디스크 상에 써진 이상한 데이타 발생 : bitmap OK, inode X, data block OK → 누가 가리키는지 모르는 경우

(4) Solutions #1 : fsck

- 초기의 파일 시스템은 Crash Inconsistent States 해결하기 위해 간단한 방법 사용
  - fsck는 Inconsistent한 상태를 consistent 상태로 만든다.
  - 기본적으로 파일 시스템이 Inconsistent States 하더라도 그대로 두었다가 rebooting 시 이를 해결

- fsck
  - 일관성 불일치를 발견하고 수정하는 Unix 도구로 파일 시스템 메타데이터들 간의 일관성을 유지 목적만 (Inode, bitmap, superblock) (Data는 관심 X Inode와 bitmap만 관심)
  - ✓ 모든 Crash Inconsistent States를 해결하지는 못함
  - ✓ 예) 아이노드가 의미 없는 데이타 블럭을 가리키고 있지만, 파일 시스템은 일관성이 있는 것처럼 보인 경우
    - Read garbage data (consistent) → 5페이지 만화 그림
  - 여러 동작으로 구성
  - fcsk 실행 시에는 파일 시스템이 어떤 동작도 실행하지 않는다고 가정

- fsck 기본 동작

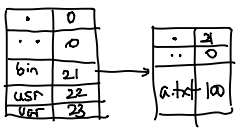
① - Super block 내용 오류 검사 (old)
  - ✓ 파일 시스템에 블럭 개수가 파일 시스템의 크기보다 더 큰지 확인 등
  - · 손상이나 문제가 있는지 의심되는 슈퍼블럭이 있는지 찾음
  - · 문제 발견 시 시스템은(또는 관리자는) 슈퍼블럭을 복사본으로 대체할지를 결정

하나의 block 사이즈가 4KB이고 block 개수가 1000개인데 Super block이는 5MB라고 써져있는 경우

② - free block 검사
  - ✓ inode block, indirect block, double indirect block 등 검사 후 정확한 할당 비트맵을 재구성
  - · 기존 비트맵과의 일치하지 않으면 inode 정보를 기반으로 비일관성 해결
  - ✓ 모든 inode 에 대해서 같은 검사를 수행
  - · inode 비트맵의 유효성을 검사하고, 필요 시 inode 비트맵 재구성

③ - inode 상태 검사
  - ✓ 각 inode 손상 여부 확인
  - · 예) inode의 유효한 속성(일반 파일, 디렉토리, 심볼릭 링크 등) 이 있는지 확인
  - ✓ 만약 inode 이 맴버 항목 중에 쉽게 해결이 불가능한 문제가 있으면 해당 inode 를 초기화하고 inode 비트맵 갱신

④ - inode link 할당된 inode 링크 개수 확인 (old)
  - ✓ inode 링크의 개수 : 특정 파일에 대한 링크를 포함하고 있는 디렉토리 수
  - ✓ 루트 디렉토리부터 모든 디렉토리 트리 탐색하여 링크 개수 수집
  - ✓ 계산된 개수와 확인된 개수가 다르면 inode의 링크 개수를 수정
  - ✓ 할당된 inode는 있는데 어떤 디렉토리도 이를 참조하지 않으면 해당 파일은 lost+found 디렉토리로 이동

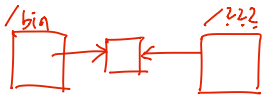


/, /bin, /usr, /var, /bin/a.txt
총 5개의 링크 수

⑤ - Duplicate 중복
  - ✓ 중복된 포인터 검사
  - ✓ 같은 데이타 블록을 가리키는 서로 다른 inode가 있는지 확인 ✓
  - ✓ 만약 한 아이노드가 문제가 있으면 초기화

⑥ - Bad block (old)
  - ✓ 모든 포인터 목록을 검사하면서 Bad block도 함께 검사
  - ✓ 어떤 포인터가 "bad"라고 판단되면 당연히 그 포인터가 유효하지 않는 공간을 참조하고 있음을 나타냄
  - · 예) 파티션 영역을 넘어서는 곳의 주소를 갖고 있는 경우
  - ✓ 아이노드나 간접블록에 해당하는 포인터 삭제(초기화)

⑦ - Directory check 디렉터리 검사
  - ✓ fsck 는 파일의 내용을 확인하는 것을 불가능하나 디렉토리 내용은 확인 가능
  - · 디렉토리 첫 항목이 "."과 ".." 인지
  - · 디렉토리의 inode가 실제 할당되어 있는지
  - · 특정 디렉토리가 두번 이상 연결되어 있는지



디렉토리는 부모와 연결될 수 X

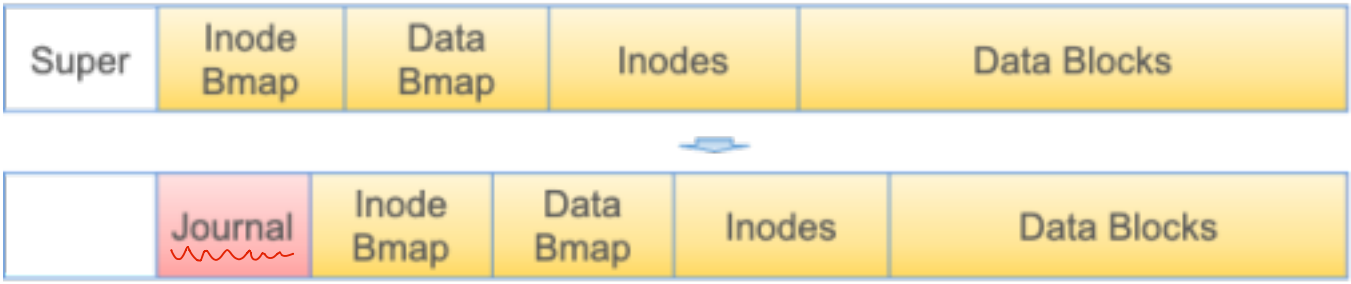
Pros	Cons
개념적으로 간단	FS에 대한 복잡한 지식 필요
별도의 writing overhead 없음	너무 느림
수행하면 FS는 consistent 해짐	Inconsistent인 경우만 해결

(5) Solutions #2 : Journaling (WAL : Write-Ahead Logging)

- Linux ext3/ext4/reiserfs/xfs, IBM JFS, SGI XFS, Windows NTFS 등이 해당
- 기본 개념

- 쓰기 요청이 오면 디스크가 즉시 갱신되지 않음
  - ✓ 새로운 페이지 이미지 전체를 저장하기도 하고, 변경될 부분만 저장 => 할 일 미리 저장 "write-ahead " + "log"
  - ✓ 해당 디스크 페이지들을 새로운 내용으로 덮어쓰기하는 과정에서 crash 발생하면, 로그를 확인하여 재갱신(redo)
- (1) 쓰기는 디스크의 저널 섹션에 메모로 기록
  - Inode가 뭔지, 어떻게 바뀌었는지
  - bitmap이 뭔지, "
  - Data이 뭔지, "

- (2) 메모에 따라 디스크 상 구조(on-disk structure) 를 덮어 씀
- (3) 덮어 쓰다가 crash가 나면, 메모를 다시 참조해서 덮어 씀



• 종류

- Data Journaling
  ✓ 디스크 업데이트에 필요한 모든 정보를 저널 공간에 기록, 동일한 작업 2번 진행 ⇒ ext3
- Metadata Journaling
  ✓ 성능을 높이기 위한 방법으로 메타 데이터만 저널링하는 방법 ⇒ ext4
  bitmap, Inode

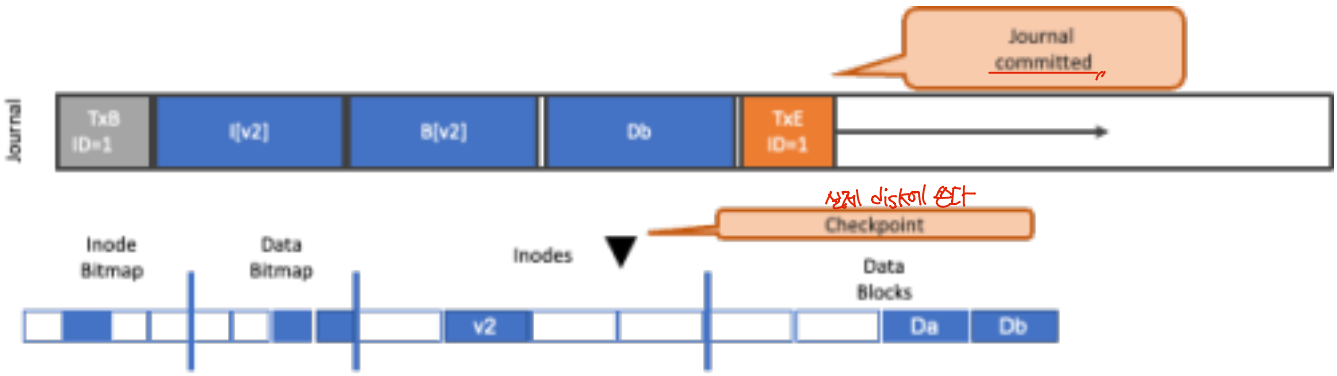
• Data Journaling (예. Linux ext3)

- I[v2], B[v2] , Db 갱신하는 평범한 파일시스템 연산 가정.
  ✓ 디스크 기록하기 전 로그 (저널) 먼저 기록 가정

단점 ⇒ Journal 용량, 느림

- 1. 저널 기록
  ✓ 트랜잭션을 저널에 기록
  ✓ 트랜잭션 시작 블럭, 갱신될 데이터 블럭와 메타데이터 블록, 트랜잭션 종료 블럭을 로그에 기록
  ✓ 이 블럭들이 디스크에 안전하게 기록될 때까지 대기
- 2. 체크포인팅 : 갱신된 메타데이터와 데이터 블럭들을 해당 위치에 반영

- Data Journaling 시나리오
  ✓ 1. Journal Write : 트랜잭션 시작 블록(TxB), 업데이트할 데이터 정보 (inode, bitmap, datablock), 트랜잭션 종료 블록 (TxE)을 저널 공간에 기록
  · TxB - 트랜잭션 begin
  ☞ 트랜잭션 identifier
  ☞ 팬딩된 갱신에 대한 정보
  · Journal 에 트랜잭션 컨텐츠 write
  ✓ 2. Journal commit
  · TxE - 트랜잭션 end
  ✓ 3. Checkpoint : 저널 공간에 있는 정보로 디스크를 업데이트 (실제 반영)
  · 지연된 메타데이타와 데이타 갱신을 write



• 저널에 기록하는 도중 crash 발생 → 해당 journal의 transaction은 완전히 없는것으로 무시됨

- 트랜 잭션(TxB, I[v2], B[v2], Db, TxE)에 속한 블럭 집합을 디스크에 쓰려고 시도할 경우

- 방법 1.

✓ 하나씩 요청을 디스크에 전달하고 각각이 완료되기를 기다렸다가 다음 기록 =〉 너무 느림

- 방법 2. ⇒ 보통 사용함

✓ 다섯 개의 요청을 한꺼번에 전송해서 디스크가 이들을 차례로 순차 쓰기

✓ 매우 효율적 ?

· 한번에 많은 양을 쓰고자 할 경우, 디스크가 스케줄링을 통해 작은 단위로 나누어 기록할 가능성 존재

· 쓰기 완료순서는 디스크에 의해 결정되며, 파일 시스템이 의도했던 순서와 다를 수 있음

· 예) 디스크 내부적으로는 (1) TxB, I[v2], B[v2] TxE 먼저 쓰고 (2) 나중에 Db를 쓸 가능성

☞ 디스크의 전원이 (1) 과 (2) 중간 끊어진다면? ⇒ 무시해버림 (transaction 무시) ≠ 데이터 날라감 → 옛 (garbage value)

메타데이터 저널링은 TxB, I[v2], B[v2] TxE만 저널에 기록하고 Db는 따로 기록

• 트랜잭션 종료 블럭 기록 =〉 트랙잭션의 성공을 의미 (트랜잭션 종료 블럭은 매우 중요)

• 잘못된 트랜잭션 종료 예

- 블럭이 저널에 기록이 되지 않았음에도 불구하고 트랜잭션이 이 성공적으로 종료된 것처럼 보임

- 시작과 끝에 동일한 식별자 순서 번호

- 비트맵이나 아이노드 블럭은 그 내용을 살펴보면 값들이 의미가 있는 값인지 알 수 없음

- 저널 트랜잭션 4번째 블럭 위치에는 파일의 데이터가 있어야 함

✓ 블럭의 내용을 읽어도 잘못된 블럭인지 아닌지를 알 수가 없음

✓ 저널의 4번째 블럭 내용으로 파일시스템이 정상인지 알 수 없음

- 시스템이 재부팅으로 복구 수행해도 문제 발생

✓ 로그 영역을 검사

· 저널에 기록된 내용들을 체크포인팅

· "??"라고 쓰여진 가비지 Db 블럭을 실제 위치에 그대로 복사

· 슈퍼블록에서 이런 일이 있어나면 더 심각 =〉 잘못 복구되면 마운트조차도 할 수 없음

- 트랜잭션 종료 블럭 기록으로 인한 문제 해결

- 파일시스템은 트랜잭션을 두 단계로 나누어 기록

✓ 1. TxE를 제외한 모든 블럭을 한 번의 쓰기 요청으로 저널에 씀

✓ 2. TxE 블럭에 대한 쓰기를 요청하여 저널을 안전하게 만듦

- 선행 조건

✓ 디스크 쓰기 연산의 원자성

· 트랜잭션 종료 블럭(TxE)는 무조건 원자적으로 기록

· 디스크는 섹터단위(512바이트) 크기의 쓰기에 대한 원자성 보장 => 512 바이트 블럭 이내 크기의 TxE (1개의 Sector)

(6) Recovery

- Crash 상황에서 저널을 사용하여 파일 시스템을 복구 방법

✓ Crash 는 파일시스템을 갱신하는 전 과정에서 발생 가능

✓ 트랜잭션이 로그에 안전하게 기록되기 전에 크래시가 발생한다면

· 즉, 위에서 말한 2단계가 끝나기 전에 할 일은 간단

· 복구 시에 아무일도 안하면 됨. 트랙잭션이 로그에 기록되지만, 체크포인트가 완료되기 전에 크래시가 발생한다면 파일 시스템의 복구 프로세스는 시스템이 부팅할 때 로그를 탐색해서 디스크에 커밋된 트랜젝션이 있는지 파악하여 복구

✓ 트랜잭션이 로그에 안전하게 기록되기 전 (journal commit 전) crash 복구 시 아무것도 하지 않으면 됨 (undo)

- 트랙잭션이 로그에 기록되었으나, 체크포인트가 완료되기(journal commit 바로 후)

✓ 저널에 commit된 트랜잰션을 replay하여 디스크 자료 구조 간에 일관성을 보장

· Commit된 트랜잭션의 블럭들을 디스크 상의 원래의 위치 쓰고 이 과정을 재생 => replayed

· 가장 간단한 방식의 로깅 => redo logging

✓ 복구 후 파일 시스템을 마운트하여 새로운 요청을 받을 수 있도록 준비

(7) Batching Log Updates

- 방법 1.

- Linux ext3 등은 여러 개의 저널로그를 모아서 한 번에 디스크에 commit 하는 방법 사용

✓ 로깅 해야 할 모든 파일시스템 갱신 내용을 transaction buffer (= journal buffer) 자료 구조에 보관

· 파일 시스템 파티션 마다 하나만 존재하는 전역 자료구조

✓ 파일시스템은 메모리에 있는 갱신 내용들을 정기적으로 저널에 기록

· 예) 5초, 저널 timeout

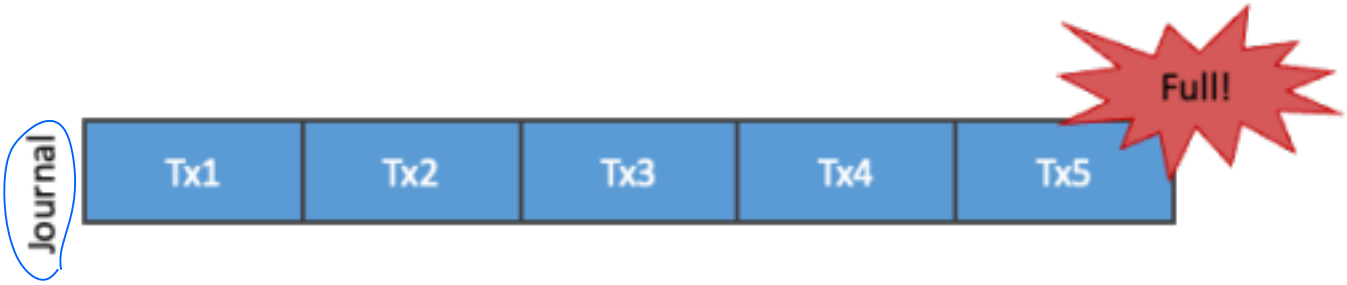
- 방법 2

- fsync()/ sync() 호출 시 트랜잭션 버퍼의 내용들이 저널에 commit

(8) Log 공간의 관리 (making the log finite)

- 로그 공간이 full 되면

- 로그에 있는 모든 트랜젝션을 (순서대로) 재실행해야 하기 때문에 로그가 커질수록 복구 소요 시간은 길어짐
- 만약 로그가 가득차면 디스크에 더 이상의 트랜잭션 commit 불가능



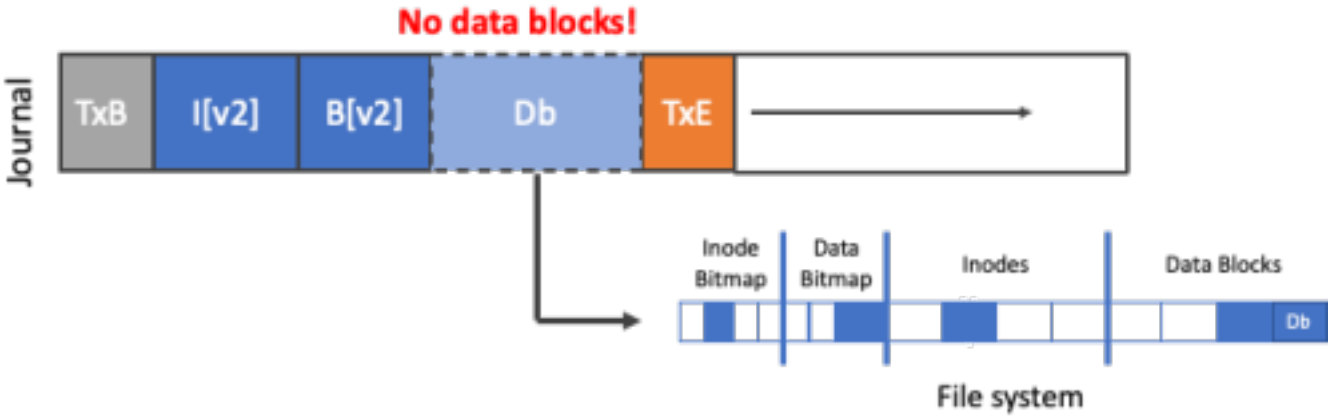
- 해결책

- Circular log 사용
  - ✓ 로그영역을 끝까지 다 쓰면, 앞에서부터 다시 쓰기
    - · 트랜잰션이 체크포인트되면 해당 로그 공간이 재사용될 수 있도록 해당 공간 비움
    - · 예를 들어 최신 트랜젝션과 가장 오래된 트랜젝션의 위치를 저널 슈퍼블럭 에 기록
    - · 두 영역 사이의 블럭들은 의미있는 로그 정보를 가지고 있으며, 그 외의 공간은 재사용 가능한 빈 공간

(9) Metadata Journaling

- 데이타 저널링(ext3)의 속도 문제 => Ordered journaling (메타데이터 저널링, XFS)

- 저널에 데이터 블럭을 기록하지 않는다는 것외 데이타 저널링과 동일
  - ✓ 데이타 저널링에서는 데이터 블럭 (Db가) 로그에 기록
  - ✓ 메타데이타 저널링에서는 추가적인 쓰기를 피하기 위해 파일 시스템의 원래 위치에 Db를 기록
  - ✓ 디스크로 내려가는 I/O 트래픽의 대부분이 데이터인 것을 고려하면 두번 쓰지 않는 것으로만으도 I/O 오버헤드 줄임



(참고) Chapter 43. Log strucutured FS

(1) Background

- John Ousterhout 와 그의 제자 Mendel Rosenlum (Stanford Univ)
- 개발 동기

- 메모리 크기가 증가 추세

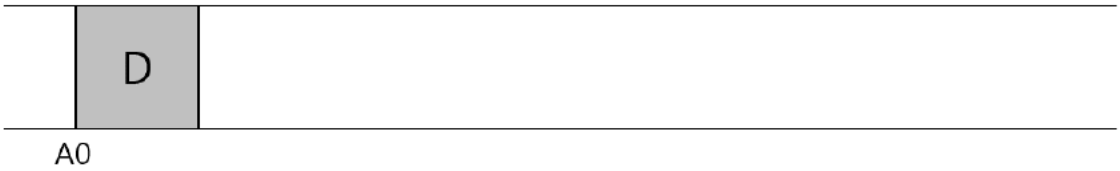
- Sequential IO의 성능이 Random IO 보다 성능이 더 좋음
  ✓ 디스크를 순차적으로 사용할 수 있다면 상당한 성능 효과

- 일반적인 워크로드에서 기존 파일 시스템들의 성능이 좋지 않음
  ✓ FFS는 블럭 하나 크기의 새로운 파일을 생성하기 위해 여러 번의 쓰기
  · 새로운 아이노드를 위해 한 번,
  · 아이노드 비트맵을 위해 한 번,
  · 파일이 들어 있는 디렉터리의 데이터 블럭을 위해 한 번
  · 디렉터리 아이노드를 갱신하기 위해 한번
  · 새 파일의 부분이 될 데이터 블럭을 위해 한 번
  · 할당된 데이터 블럭을 표시하기 위해 데이타 비트맵을 위해 한 번,

- 기존 파일 시스템은 RAID를 고려하지 않음
  · RAID-4와 RAID-5는 작은 크기의 쓰기(small write) 문제가 있음
  · 하나의 블럭을 쓰더라도 실질적으로 4번의 물리적 I/O가 발생

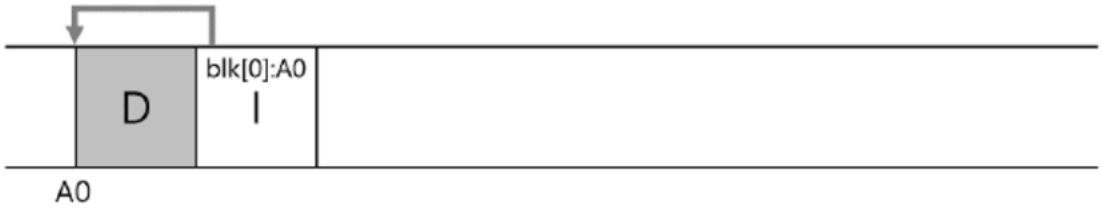
(2) 디스크에 순차적으로 쓰기

- 파일 시스템을 변경하는 모든 쓰기 작업을 어떻게 순차 쓰기로 디스크에 보낼 수가 있을까?
- 파일에 데이터 블럭 D를 쓴다고 가정 : 디스크의 주소 A0에 D를 씀



- 데이터 블럭을 쓰면

- 데이터 갱신 외 메타데이타 갱신도 필요
  ✓ 해당 파일의 아이노드 (I) 갱신, 아이노드가 데이터 블럭 D를 가리키도록 해야 함



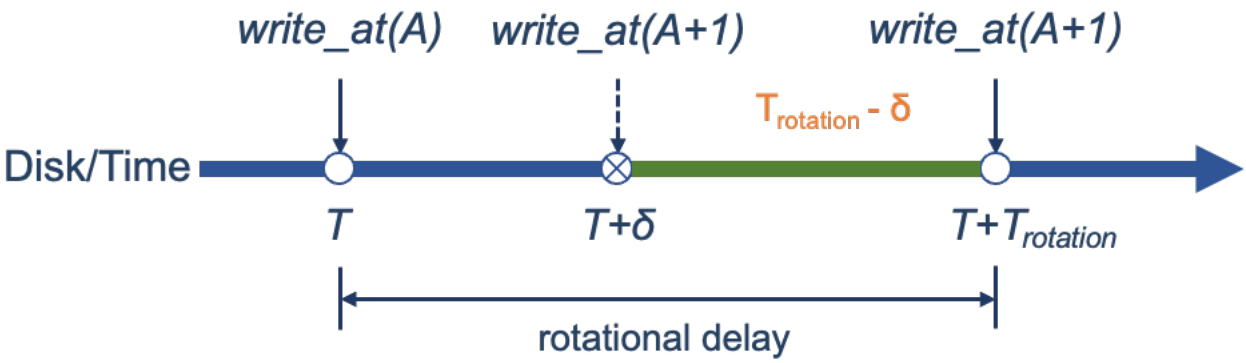
(3) 순차적이면서 효율적으로 쓰기

- 순차 쓰기가 효율적인가?

- 첫번째 쓰기(write_at(A) 와 두 번째 쓰기 (write_at(A+1) 사이에 디스크가 회전
  ✓ 두번째 쓰기를 하기 전에 플래터가 회전되고 회전 시간이 Trotation 시간이 걸린다면, 디스크 표면에 두 번째 쓰기를 실행하

기 전에 Trotation – δ 시간을 기다려야 함

- 순차 쓰기는 디스크의 최대 성능을 달성하지 못함 =〉 다수의 순차쓰기 (또는 한번에 큰 크기 쓰기) 필요



– 한 번에 디스크에 내려 보내야 빠른 쓰기 성능 가능

- LFS =〉 쓰기 버퍼링 기법 사용

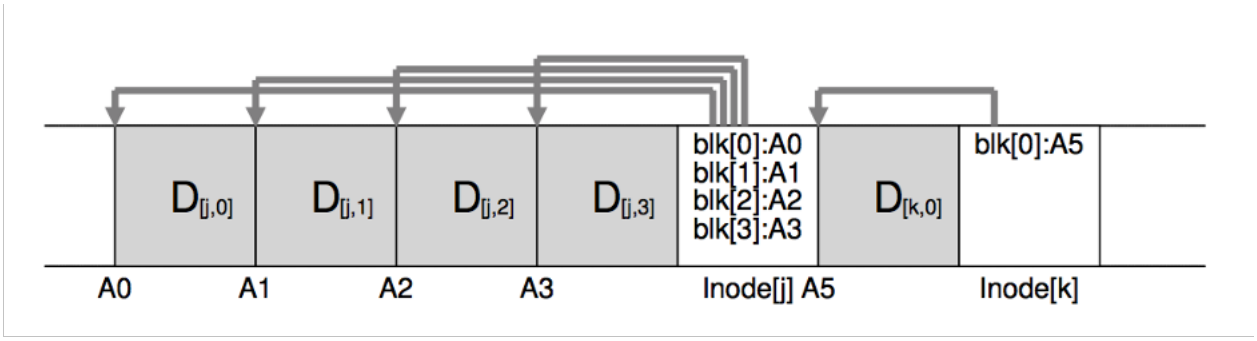
– 디스크에 쓰기 전에 갱신 내용 메모리(세그먼트 버퍼)에 보관

– 갱신 내용이 충분히 쌓이면 디스크로 한번에 모두 보냄

– Segment : 디스크에 한번에 기록하는 단위 =〉 Segment가 충분히 크면 쓰기 연산은 효율적

– 모든 갱신(데이터 블럭과 아이노드 등)을 디스크에 순차적으로 기록

- 두개의 갱신 내용을 하나의 세그먼트 버퍼에 넣는 예



✓ 첫번째 갱신 : 파일 j (inode[j])에 4개의 블럭을 쓰는 것

✓ 두번째 갱신 : 파일 k (inode[k])에 1개의 블럭을 쓰는 것

- LFS는 총 7개의 블럭으로 이루어진 세그먼트를 디스크에 한번에 씀

(4) How Much To Buffer?

- 디스크에 쓰기 전에 LFS 는 몇개의 쓰기를 버퍼에 가지고 있어야 하는가?

– 디스크의 물리적 특성(전송속도 대비 위치 선정 오버헤드 – 회전과 탐색 오버헤드)마다 다름

– 위치 선정 비용 (디스크 헤드 이동 시간)을 상쇄시키기 위한 버퍼의 크기

✓ 클수록 좋음

✓ Dmb의 크기를 쓴다고 가정할 때

· 디스크 헤드를 옮기는 시간 : Tposition

· 디스크 전송 속도 : Rpeak MB/s

· 데이타 청크를 쓰는데 걸리는 시간 : Twrite = Tposition + D/Rpeak

· 실제 쓰기 속도 (유효 속도) : Reffective = D/Twrite = D/(Tposition + D/Rpeak)

· 최대 속도에 근접하는 유효 속도

☞ 유효 속도가 최대 속도의 특정 비율(F)이 되게 하는 것

☞ 0 〈 F 〈 1
☞ 일반적으로 F =0.9 (90%)
☞ Reffective = F * Rpeak
☞ Reffective = D/(Tposition + D/Rpeak) F * Rpeak
☞ D = F * Rpeak * (Tposition + D/Rpeak)
☞ D = (F * Rpeak * Tposition) + (F * Rpeak * D/Rpeak)

- $D = F/(1-F) * Rpeak * Tposition$

$$T_{write} = T_{position} + \frac{D}{R_{peak}}$$

Time to write

Size of data (MB)

Disk transfer rate (MB/s)

$$R_{effective} = \frac{D}{T_{write}} = \frac{D}{T_{position} + \frac{D}{R_{peak}}}$$

Effective rate of writing

Fraction of the peak rate ($0 < F < 1$)

$$R_{effective} = \frac{D}{T_{position} + \frac{D}{R_{peak}}} = F \times R_{peak}$$

$$D = \frac{F}{1-F} \times R_{peak} \times T_{position}$$

- Example

- 디스크 헤드 이동 시간 (위치 선정) : 10 ms, $T\ position = 0.01s$
- 최대 전송 속도 : 100MB/s, $R\ peak = 100MB/s$
- 유효 대역폭 : 90%, F=0.9
- D = 0.9/0.1 * 100MB/s * 0.01 sec. = 9MB

- 최대 대역폭에 도달하기 위해 버퍼에 얼마나 저장해야 할지 ?
- 95%와 99%가 되려면 얼마나 되어야 할까?

(5) Find inode

- 일반적인 UNIX FS에서 inode 찾기

- 정해진 위치에 배열로 배치 -〉 찾기가 간단
- 전통 파일 시스템에는 각 inode 위치가 정해져 있음
  ✓ inode 번호 * inode 크기 + 배열의 시작 주소

- FFS에서 inode 찾기

- FFS는 inode 테이블을 분할하여 실린더 그룹마다 inode 노드 블럭이 따로 있음 -〉 복잡
- 분할된 테이블(group)의 크기와 각각의 시작 주소를 알아야 함

- LFS에서 inode 찾기

- inode 디스크 전역에 흩어져 있음 -〉 더 복잡
- 원래 위치에 덮어쓰기를 하지 않기 때문에, 최신 inode 의 위치가 변함

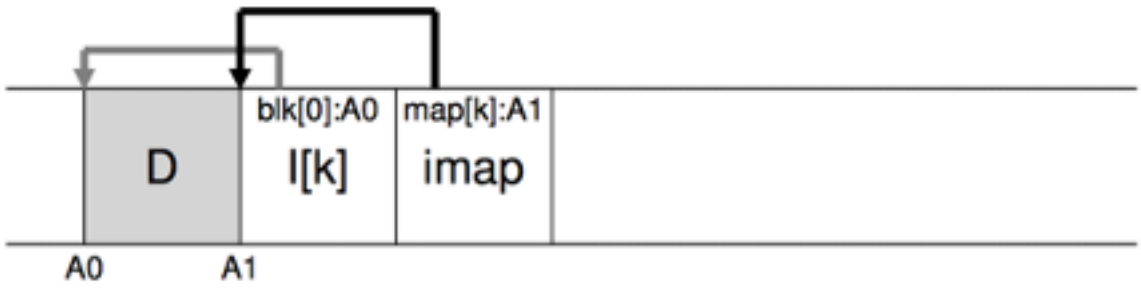
(6) The inode Map in LFS

- Where should the imap reside on disk

- inode map(imap) 자료 구조
  - ✓ imap 자료 구조는 inode 번호를 입력으로 하여 가장 최신 inode 의 디스크 위치 구함
  - ✓ imap 구조는 각 항목당 4바이트 (디스크 포인터) 를 갖는 간단한 배열로 구현
- 디스크에 imap 가 기록될 때 imap은 새로운 위치를 가리키도록 갱신

- imap의 보관은 안전하게 =〉 디스크에 써져야 함

- LFS에 crash가 발생하더라도 inode의 위치 파악 가능
- 저장 위치
  - ✓ 디스크의 고정된 위치에
  - ✓ LFS는 inode 맵을 새로이 기록된 데이터와 inode 들 옆에 함께 기록
  - ✓ 파일 k에 데이터 블럭을 추가 시 새로운 데이터 블럭과 해당 inode 그리고 impa의 일부분을 연속하여 디스크에 기록



- imap 을 찾는 방법

- imap도 여러 블럭으로 나누어져 디스크 상에 흩어져 있게 됨
- 특별한 방법 없음

- LFS에서 imap 을 찾는 방법

- 디스크 상에 약속된 위치에 각 imap 블럭들의 위치 기록
  - ✓ 체크포인트 영역(checkpoint region, CR)
  - ✓ 최신의 imap을 이루는 블럭들을 가리키는 포인터를 가지고 있음 =〉 CR 영역을 읽어 imap 조각을 찾을 수 있음
  - ✓ CR은 주기적(약 30초)으로 갱신 =〉 성능에 큰 영향은 없음
  - ✓ 디스크 상의 파일 시스템 영역에 CR 영역이 할당되어 있음
  - ✓ UNIX 파일 시스템과 같이 imap 조각들은 inode의 주소들을 포함하며, inode는 파일과 디렉토리를 가르킴