Crash Consistency: FSCK and Journaling

2017.07.17

DC Lab

서영근

Content

1. Introduction
2. A detailed example
    1. Crash Scenarios
3. Solution 1: The file system checker
4. Solution 2: Journaling(or Write-Ahead Logging)

Introduction

- **다른 Data structure와 다르게, file system data structure 는 '일정'해야 한다.**
  - ✓ '일정'의 의미는 전원이 나가거나 crash가 나도 계속 유지할 수 있다는 것.
  - ✓ 이는 file system에서 가장 중요한 문제

- **이를 가능하게 하는 두 가지 방법**
  - ✓ Fsck (file system checker)
  - ✓ Journaling (write-ahead logging)

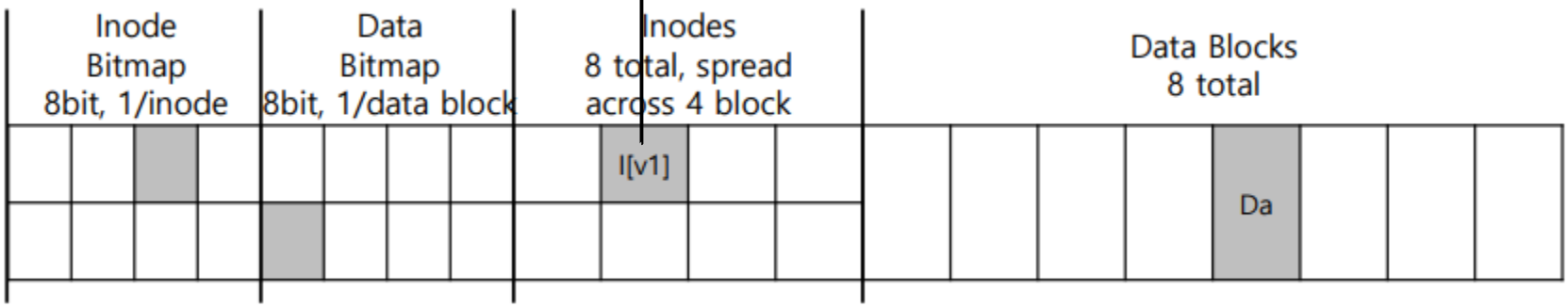
A detailed example

-> file size 는 1(Da)
-> file pointer 는 4
-> data block 4에 data가 있다는 뜻
-> 다른 포인터는 null
-> 더 이상 데이터가 없다.

◆ **Workload 설정 (single data block 의 확장)**

- ✓ file open
- ✓ lseek() 를 써서 파일 끝으로 이동
- ✓ 4KB write
- ✓ file close

```
owner           : remzi
permissions     : read-only
size            : 1
pointer         : 4
pointer         : null
pointer         : null
pointer         : null
```

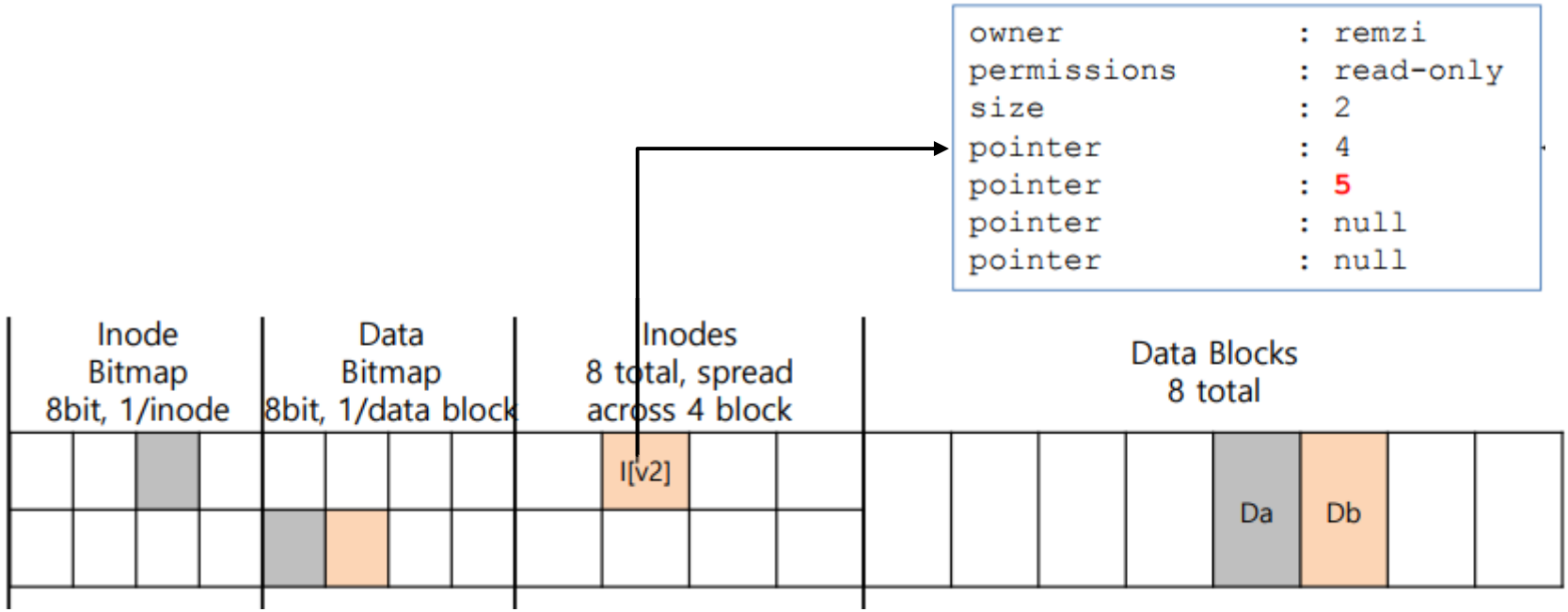


◆ **Data block 확장 전 상태**

- ✓ Inode Bitmap(00100000), Data Bitmap(00001000),
- ✓ Inode number 2에 버전 1, Data block 4에 Da

A detailed example

-> file size 는 2(Da, Db)
-> file pointer 는 4, 5
-> data block 4, 5에 data가 있다
-> 다른 포인터는 null
-> 더 이상 데이터가 없다.



◆ Data block 확장 후 상태

✓ Inode Bitmap(00100000), Data Bitmap(00001100),

✓ Inode number 2에 **버전 2**, Data block 4에 Da, **Data block 5에 Db**

A detailed example

◆ **single data block 의 확장의 결과**

- ✓ Data Bitmap, Inode 버전, Data block datas -> 총 3군데의 write가 필요하다.

◆ **Write() syscall 시 write는 바로 일어나지 않는다.**

- ✓ Data Bitmap, Inode 버전, Data block datas 는 처음에 main memory 에 위치할 것이다. (page cache 혹은 buffer cache에)
- ✓ 일정 시간 이후 디스크에 write 가 일어나는 것
- ✓ 3가지 write 중에 crash가 난다면 file system consistency에 영향을 주는 것

Crash Scenarios (write 1개만 성공)

◆ **1. Data 만 write 됐을 경우**

✓ 이 경우에는 file system consistency 에 전혀 영향이 없다. 왜냐하면 inode도 데이터를 가리키지 않고, bitmap도 할당정보가 없기 때문이다.

◆ **2. inode 만 write 됐을 경우**

✓ Write 된 inode 값을 믿을 경우, 쓰레기 값을 읽어오게 된다.

✓ 또한 bitmap은 할당 되지 않았다고 하는데 inode는 block을 가리키게 되 data inconsistency 가 발생한다.

◆ **3. data bitmap 만 write 됐을 경우**

✓ 위와 같은 원리로 data inconsistency 가 발생한다.

✓ 추가적으로 이 상태에서는 해당되는 block을 영원히 쓸 수 없는 space leak 현상이 발생한다.

Crash Scenarios (write 2개 성공)

◆ **1. Data 만 write가 안됐을 경우**

✓ 이 경우에는 file system consistency 에 전혀 영향이 없다. 그러나 문제의 block은 쓰레기 값을 가지게 된다.

◆ **2. data bitmap 만 write가 안됐을 경우**

✓ bitmap은 할당 되지 않았다고 하는데 inode는 block을 가리키게 되 data inconsistency 가 발생한다.

◆ **3. inode 만 write 됐을 경우**

✓ 위와 같은 원리로 data inconsistency 가 발생한다.

✓ Inode가 write 되지 않아서 가리키는 포인터가 없어 데이터에 접근 할 수 없다.

Solution 1: The file system checker

◆ **간단한 접근**

- ✓ inconsistency 가 발생하도록 내버려 두고, 이후에 문제를 해결하는 방식
- ✓ 이러한 방식은 모든 문제를 해결하는 것은 불가능 하다.
- ✓ Ex) block data 만 write를 실패한 경우 -> consistency를 유지하고 있는 것처럼 보여서 해결이 불가능

◆ **동작원리**

- ✓ Filesystem이 mount 되고 사용가능해지기 전에 fsck가 작동
- ✓ file system이 consistence 하게 만들고 작동을 멈춘다.

Solution 1: The file system checker

◆ 1. super block 체크

- ✓ sanity check
- ✓ Ex) filesystem size가 할당된 블록 수 보다 큰가?
- ✓ 데이터의 이상이 있는 것 같은 super block을 찾는 것이 목표

◆ 2. Free block 체크

- ✓ 파일 시스템의 구조를 만들기 위한 것
- ✓ Inode, indirect block, double indirect block 등을 스캔한다.
- ✓ Bitmap 과 inode 사이의 inconsistency 가 있는지 조사
- ✓ Inconsistency가 있다면 inode의 정보를 기준으로 수정

Solution 1: The file system checker

◆ 3. inode state 체크

- ✓ inode가 훼손된 상태인지 검사
- ✓ Regular file, directory, symbolic link등을 조사한다.
- ✓ 이상이 있는 것 같은 inode는 지워진다.

◆ 4. inode links 체크

- ✓ Inode 들의 link count 를 확인한다.
- ✓ Root 부터 차근차근 돌면서 전체 파일시스템의 파일과 디렉토리 링크를 계산한다.
- ✓ Inode 값과 다른게 있으면 inode를 수정

Solution 1: The file system checker

◆ 5. Duplicate 체크

- ✓ 두개의 다른 inode가 같은 block을 가르키는지, 즉 중복 포인트를 체크한다.
- ✓ 둘 중에 하나가 확실히 잘못된거면 삭제한다.
- ✓ 그렇지 않다면 block은 복사되어 각각의 inode는 복사된 block을 하나씩 pointing 하게 된다.

◆ 6. bad block 체크

- ✓ Bad block은 유효한 범위를 명백하게 넘어가는 block이다.
- ✓ 예를 들면 파티션 size 보다 더 큰 블록을 참조하는 경우가 있다.
- ✓ 이때, fsck는 inode의 포인터를 지우는 것만 수행한다. (일종의 판단이 없다.)

Solution 1: The file system checker

◆ 7. Directory 체크

- ✓ file은 특정 포맷이 없지만 디렉토리는 특정한 포맷 정보가 존재한다.
- ✓ Fsck는 디렉토리에 대해 이러한 특정한 포맷 정보를 검사한다.
- ✓ 예를 들어, ./ ../ 같은 것들이 첫번째 엔트리 인지 등이 있다.

◆ FSCK 의 단점

- ✓ fsck의 동작이 매우 여러가지 이기 때문에 구현이 어렵다.
- ✓ 결정적으로, fsck는 모들 블록을 스캔하기 때문에 **매우 느리다.**
- ✓ 이는 매우 중요한 문제이다.
- ✓ 왜냐하면 3번의 write를 하는 하나의 트랜잭션을 찾기위해 전체 파일시스템을 스캔하는 것은 매우 비효율적이기 때문이다.

Solution 2: Journaling

◆ 아이디어

- ✓ 기본적인 아이디어는 DBMS의 write-ahead logging에서 나왔다.
- ✓ 업데이트 도중에 crash가 나면 했던 작업을 다시 할 수 있는 걸 게런티 해야한다.
- ✓ 따라서 업데이트 하기 전에(write-ahead) 무엇을 할 것인지 logging 한다.
- ✓ 이것이 write-ahead logging, 즉 journaling의 기초 개념

◆ linux ext3

- ✓ 전체적인 구조는 ext2 와 매우 유사하다.
- ✓ 디스크를 block group 들로 나뉘어 지고, 각 block group 은 inode bitmap 와 data bitmap 들을 가지고 잇고 더하여, inode 들과 data block 들을 가지고 있다.
- ✓ 여기에 journal part가 추가 된 것이 ext3의 구조이다.

Data Journaling

◆ Data Journaling

✓ Linux ext3 에서 data journaling이 가능하다.

✓ inode, block bitmap, block data를 디스크에 쓰기 전에 저널에 먼저 쓴다.

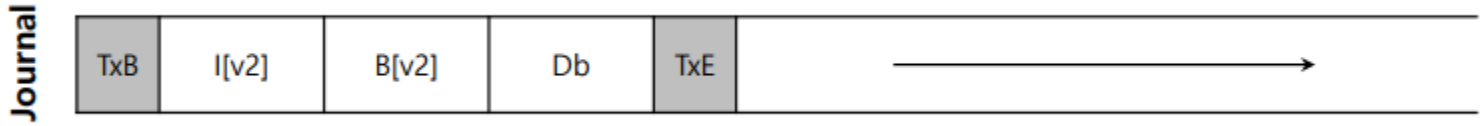
◆ journal의 구조

✓ inode, block bitmap, block data 는 기본적으로 들어간다.

✓ 추가적으로, TxB : 업데이트의 시작을 알림 transaction identifier (TID)

//TxE : 업데이트의 끝 transaction identifier (TID) 의 정보가 들어간다.

✓ 그림으로 구조를 확인해 보면,



✓ 이런 식으로 TxB 와 TxE가 저장하고 싶은 데이터를 감싸는 구조이다.

Data Journaling

◆ physical logging vs logical logging

- ✓ 지금 예시로 들고 있는 것이 전자이다. 실제 데이터가 같이 logging 되는 것
- ✓ Logical logging은 "이 업데이트는 파일 X에 데이터 블럭 Db 를 확장하려고 한다" 는 식의 묘사를 업데이트 하는 것 -> 퍼포먼스의 증가 효과가 있다.

◆ checkpointing

- ✓ logging, journaling이 끝나면 checkpointing이 시작된다.
- ✓ 실제 디스크 위치에 값을 write 하는 것

Data Journaling

1. journal write

- TxB를 포함한 transaction 을 로그에 쓰고, 모든 pending data 와 metadata 를 로그에 업데이트 하고, TxE 를 로그에 쓴다. 이 쓰기가 완료되는 것을 기다린다.

2. checkpoint

- 미루어진 metadata 와 data update 들을 filesystem 의 그들의 최종위치에 write 한다.

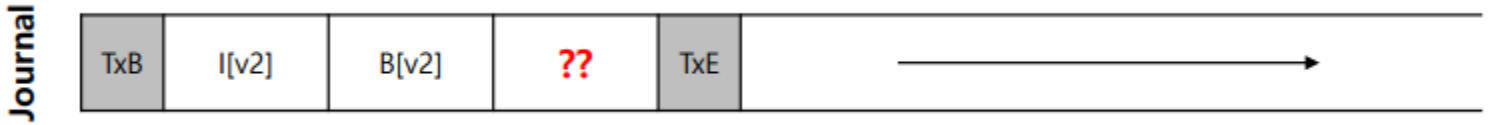
Data Journaling

◆ journal write 도중 crash 문제

- ✓ 이는 5가지 write를 한가지씩 안전하게 수행 하면서 해결할 수 있다.
- ✓ 그러나 이러한 방법은 속도가 매우 느리다.
- ✓ 따라서 우리는 위험하지만 5가지 쓰기를 동시에 수행해야 한다.

◆ 5가지 동시 쓰기의 문제

- ✓ disk 내부적으로 스케쥴링을 하는 것에서 발생하는 문제
- ✓ 예를 들어, Data를 가장 마지막으로 write하는 스케쥴을 짰다고 가정하자.
- ✓ 그리고 데이터를 쓰기 직전 crash가 났다고 가정하자
- ✓ 이 상태에서 file system은 data 값에 오류가 있는지 없는지 알지 못한다.
- ✓ 만약 저 데이터 값이 super블락과 같은 중요한 내용일 경우 시스템에 치명적일 수 있다.



Data Journaling

◆ 해결 방안

✓ (TxB, inode, bitmap, data) 와 (TxE) write 과정을 구분 짓자. 그렇다면 3가지 데이터를 가져오는데 실패하는 경우는 사라진다.

1. journal write

- TxB를 포함한 transaction 을 로그에 쓰고, 모든 pending data 와 metadata 를 로그에 업데이트 한다.

2. journal commit

- transaction commit block(TxE 를 포함하는) 을 log 에 쓰고, 완료되는 것을 기다린다. 이후에 transaction 은 committed 되었다.

3. checkpoint

- 미루어진 metadata 와 data update 들을 filesystem 의 그들의 최종위치에 write 한다.

Recovery

◆ **크래쉬가 TxE 가 쓰기 전에 발생하면**

- ✓ Skip

◆ **checkpoint 에서 crash가 발생**

- ✓ 시스템이 부팅할때, file system recovery 프로세스가 로그를 스캔하고 디스크에서 commit된 transaction을 살펴본다.
- ✓ 이 transaction은 순서대로 replay 된다. 그리고 이 block 들은 그들의 최종 디스크 위치에 write 를 시도한다.
- ✓ 이 logging 방식은 가장 단순한 방식 중에 하나이고, redo logging 이라고 부른다.

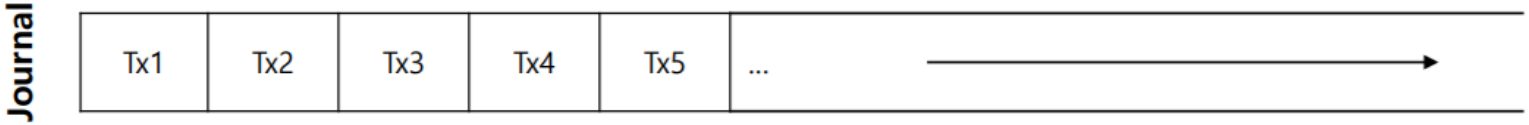
Batching Log Updates

- **Journal log commit 의 일괄처리 필요성**
  - file1 과 file 2가 같은 디렉토리 내에서 연속적으로 생성 된다고 가정해보자
  - 두 파일은 동일한 디렉토리 내 파일이기 때문에 같은 inode block 안의 inode 값을 가질 것이고, 이는 불필요한 write를 늘린다.

- **해결방안**
  - 매번 디스크 업데이트 마다 commit을 하는 것이 아닌 일정한 timeout 때마다 모든 업데이트를 한꺼번에 commit 할 수 있다.
  - 이런 방법을 이용하면 disk 쓰기 트래픽을 줄일 수 있다.

Making The Log Finite

- **Journal Space full 문제**
  - 로그는 고정된 크기를 가지고 있기 때문에 트랜잭션을 계속 추가한다면 journal space는 꽉차고 말 것이다. 이로 인해 2가지 문제가 발생한다.
  - 1. 리커버리의 시간이 길어진다. (약간의 문제)
  - 2. 더이상 트랜잭션의 커밋이 불가능하다. (심각한 문제)
  - 2번의 문제를 해결하기 위해 반드시 이 문제를 해결해야 한다.

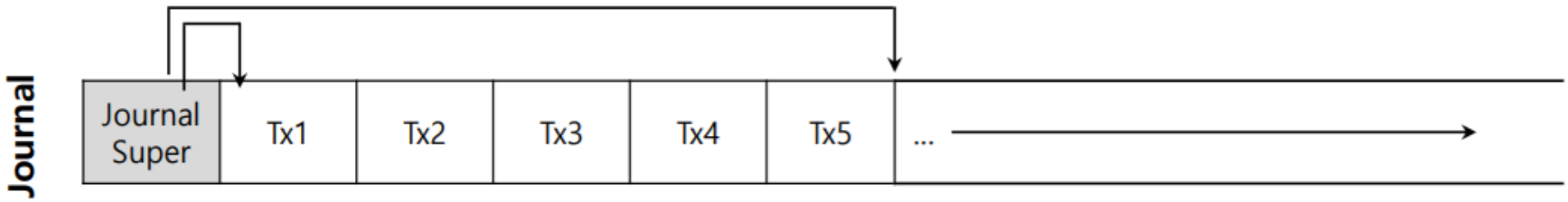


- **해결방안**
  - Circular log 즉, 저널 공간이 재활용 될 수 있어야 한다.

Making The Log Finite

- Circular log journal
  - 이를 구현하기 위해선 journal 공간에 몇가지 수정이 필요하다.
  - 우선, checkpoint 이후에 journal 공간을 free 해주는 것이 필요하다.
  - 이를 위해서 journal super block을 두어 관리하는 방법이 있다.

- journal super block
  - Checkpoint 이후 journal super block이 oldest newest 트랜잭션 포인터를 수정하므로 써, journal 공간을 free 해주는 것을 구현할 수 있다.



Making The Log Finite

1. journal write

- TxB를 포함한 transaction 을 로그에 쓰고, 모든 pending data 와 metadata 를 로그에 업데이트 한다.

2. journal commit

- transaction commit block(TxE 를 포함하는) 을 log 에 쓰고, 완료되는 것을 기다린다. 이후에 transaction 은 committed 되었다.

3. checkpoint

- 미루어진 metadata 와 data update 들을 filesystem 의 그들의 최종위치에 write 한다.

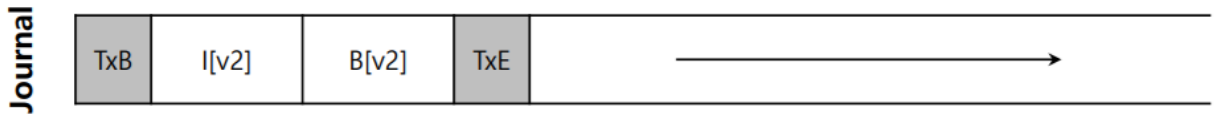
4. free

- 얼마후에, journal superblock 의 갱신에 의해journal 안에 transaction free 가 mark 된다.

Metadata Journaling

- **아직도 남아있는 문제**
  - 데이타 블록을 2번 쓰는 오버헤드가 존재한다.
  - 디스크에 쓰는 양이 2배 늘어난 것은 성능에 매우 치명적이다.

- **metadata journaling**
  - Ordered journaling 이라고도 하며, data block을 제외한 metadata(inode, bitmap) 만을 journal 하는 시스템이다.



Metadata Journaling

- **Data block 쓰기 문제**
  - data block을 journal 하지 않는 방법에서 data 블록을 언제 disk 에 write 해야 될 지에 관한 문제
  - 1. journaling 한 후 : consistency는 유지되지만 inode가 쓰레기 값을 가리키고 있는 상황이 발생한다.
  - 2. journaling 하기 전 : 이 때가 맞는 타이밍 이다.

Metadata Journaling

1. datawrite

- 최종 파일시스템 위치에data를 write한다.

2. journal metadata write

- TxB를 포함한 metadata 를 로그에 업데이트 한다.

3. journal commit

- transaction commit block(TxE 를 포함하는) 을 log 에 쓰고, 완료되는 것을 기다린다. 이후에 transaction 은 committed 되었다.

4. checkpoint

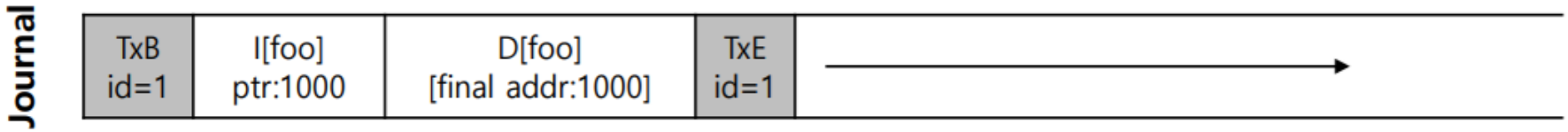
- 미루어진 metadata 와 data update 들을 filesystem 의 그들의 최종위치에 write 한다.

5. free

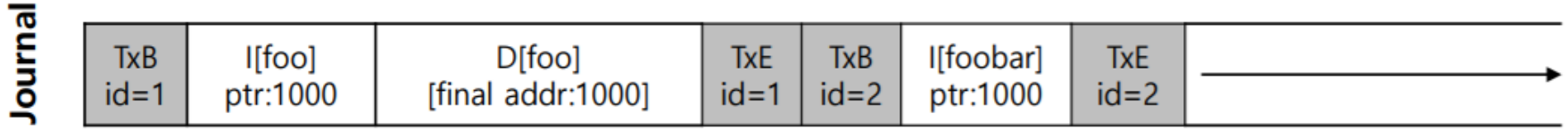
- 얼마후에, journal superblock 의 갱신에 의해journal 안에 transaction free 가 mark 된다.

Tricky Case: Block Reuse

- **Block reuse 문제 (예시로설명)**
  - 1. 디렉토리 foo를 만든다. 디렉토리 데이타는 메타데이타라서 저널링에 들어감

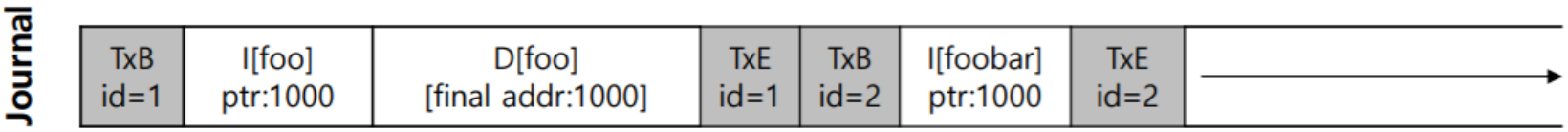


  - 2. 이후 foo 디렉토리를 지우고 같은 데이터 블럭에 foobar 라는 파일을 만든다.



Tricky Case: Block Reuse

- Block reuse 문제
  - 이 상태에서 crash가 나서 recovery가 실행 될 경우, foobar file에 foo 데이타가 들어가 버리게 되는 상황이 발생한다.



- Solution
  - Ext3에 적용된 방법인데 revoke record 라는 새로운 타입의 레코드를 만든다.
  - 지워진 파일에 대해선 revoke record를 적용한다.
  - 이후에 recovery과정에서 revoke record 에 관해서 replay를 하지 않는 다면 이 문제는 해결된다.

감사합니다