http://pages.cs.wisc.edu/~remzi/OSTEP/

Crash Consistency: FSCK and Journaling

문서에 대한 해석.

문서 링크 : http://pages.cs.wisc.edu/~remzi/OSTEP/file-journaling.pdf

42.1 A Detailed Example

test 의 workload 는 단순하다고 가정한다. / 존제하는 파일의 single data block 의 확장

append 는1. file open
2. lseek 로 파일 끝으로 offset 이동
3. 4KB write
4. file close

inode bitmap : 8 bitdata bitmap : 8 bit
inodes : 8개(0~7), 4개의 블럭에 흩어져 있음
data block : 8개(0~7)

- inode num 2
1개
v1(수정되지 않은 첫 버전)
I[v1] 으로 명명
- data block
num 4
1개
Da 로 명명

파일을 append 할때 수정되는 부분
I[v2] : inode 의 세 필드 수정
size
point[0]
point[1]

Db : 새로운 data block
B[v2] : data block 이 새로 할당 되었으므로 Data Bitmap 업데이트

이 트렌젝션의 실행을 위해서, 파일 시스템은 inode(I[v2]), bitmap (B[v2]) 그리고, data block(Db)인 3개의 분리된 write 를 해야 한다.

보통, 이러한 write 는 user 가 write() system call 을 호출하였을대 바로 일어나지 않는다는 것을 주목하라.
더 정확히 말하면, 처음에는, dirty idnoe, bitmap, data 는 main memory 에 위치할 것이다. (page cache 혹은 buffer cache에)
그리고 나서 5~30초 후에 파일 시스템이 마지막으로 디스크에 쓰기를 결정 할때, 디스크에 쓰기를 할 것이다.

운나쁘게 crash 가 발생한다면 그것은 디스크 update 에 영향을 미친다. 특히, crash 가 하나 혹은 두개의 write 이후에 발생한다면, 이 파일 시스템은 웃긴 상태로 남겨지게 될 것이다.

< Crash Scenarios >

이 문제를 좀 더 이해하기 위해, crash 상황을 예로 들어 보자.
단 하나의 write 만 성공했다고 상상해 보자. 3가지의 결과가 있다. 여기 리스트 해본다.

1. Db 만 디스크에 쓴 경우

이 경우 데이터는 디스크에 있으나, 이를 가르키는 inode 가 없고, 이 block 이 할당되었다고 가르키는 bitmap 도 없다. 따라서, 쓰기가 일어나지 않은것과 같은 상태이다. 이 경우에는 파일시스템의 crash consistency관점에서 문제가 전혀 없다.

2. inode(I[v2]) 만 디스크에 쓰여진 경우

이 경우에 i node 는 Db 가 쓰여진 disk address (5) 를 가르키고 있다. 하지만, Db 는 아직 거기에 쓰여지지 않았다. 따라서 우리가 이 포인터를 믿는 다면, 우리는 쓰래기 데이터를 이스크로부터 읽에 된다. (disk address5 에 쓰여진 옛날 데이터)

더 나아가, 우리는 새로운 문제를 가진다. 이는 "filesystem inconsistency" 라고 부른다. disk bitmap 은 data block 5 가 할당되지 않은것으로 되어 있다. 그러나 inode 는 할당되었다고 표시한다. 이 file system data structure안의 불일치는 inconsistency 하다. file system 을 사용하기 위해서, 우리는 이 문제를 어떻게든 해결해야 한다. (추가내용이 밑에 있다.)

3. **bitmap (B[v2]) 만 디스크에 업데이트 된 경우**.
이 경우에, bitmap 은 block 5 가 할당되었다고 가르킨다. 그러나 inode 는 이를 가르키고 있지 않다. 따라서 file system 의 inconsistent 가 다시 발생한다. 만약 이것이 해결되지 않은 상태로 남아 있다면, 이 write 는 space leak 을 낳게 된다. block 5 는 file system 에 의해서 결코 사용될 수 없다.

추가적인 3개의 crash 시나리오가 있다. 이 경우에 2개의 쓰기는 성공하지만 나머지 하나는 실패한다.

1. I[v2] 와 B[v2]만 써짐. Db 실패
이경우에, file system metadata 는 완전히 consistent 하다. inode 는 block 5 를 가르키고, bitmap 은 5 가 사용되고 있다고 가르키고 있다. 따라서 파일시스템 메타데이터 관점에서는 괜찮아 보이지만, 5 blcok 에 쓰래기 데이터를 가지는 문제가 있다.

2. **I[v2]. Bb 성공, B[v2] 실패**

inode 는 맞는 디스크 데이터를 가르키고 있으나 inconsistency 문제가 있다. inode 와 old version 의 bitmap 사이에. 다라서, filesysem 을 사용하기 전에 이 문제를 해결할 필요가 있다.

3. B[v2], Db 성공, I[v2] 실패

inode 와 data bitmap 사이에 inconsistency 문제 발생한다.
block 이 write 되었고, bitmap 이 그것의 사용을 가르키고 있긴 하나, 우리는 어느 파일이 그것을 사용하는지 모른다.

The Crash Consistency Problem

희망적으로, 이 crash 시나리오로부터, 우리는 크래시 때문에 파일시스템 이미지에 생기는 많은 문제를 볼수 있다.; inconsistency, space leak, 쓰래기 데이터 등등...우리가 하고자 하는것은 filesystem 을 consistent 상태로 만드는 것이다. (예를 들면 파일이 확장되기 전상태, 혹은 inode, bitmap, data block 이 디스크에 다 쓰여진 상태). 운나쁘게, 우리는 이것을 쉽게 할 수 없다. 왜냐하면 디스크는 한번에 하나의 write 만 commit 할 수 있기 때문이다. 그리고 crashe 들이나 power loss 는 이 업데이트 사이에 일어날 수 있기 때문이다. 우리는 이 일반적인 문제를 crash-consistency proble 이나 consistent-update problem 이라고 부른다.

42.2 Solution #1: The File SystemChecker Early

예전 파일시스템은 crash consistency 를 위해 간단한 접근을 취했다. 기본적으로, 그들은 inconsistency 들이 발생하게 내버려 두고, 나중에(부팅할때) 이를 fix 하도록 결정하였다. 이 느긋한 접근의 고전적인 예는 fsck 라는 tool 에서 찾을 수 있다. fsck 는 unix tool 이다. inconsistency 를 찾고 이를 수정한다. 유사한 tool 들이 다른 시스템에 존재한다. 이 접근은 모든 문제를 해결할 수 없음을 주목하자. 예를 들면, 파일시스템은 consistent 를 보고 있지만, inode 가 쓰래기 데이터를 가르키고 있다. 진짜 해결은 파일 시스템 메타데이터가 내부적으로 consistent 하도록 확인하는 것이다.

MK96 논문을 보면 fsck 는 다수의 동작으로 작동한다. 이것은 fiesystem 이 마운트 되고 사용가능해지기 전에 동작한다. (fsck 는 그것이 동작하는 동안 동작하는 활성화된 다른 파일시스템이 없다고 상정한다.); 이것이 끝났을때 파일시스템 은 conssitent 하고 사용자에 의해 접근가능하게 만들 수 있다.

fsck 가 하는것을 요약했다.- superblcok
fsck 는 슈퍼블럭이 타당한지를 먼저 check 한다. 거의 대부분이 온전한 생태 check 를 한다. 예를 들어 filesystem size 가 할당된 블럭 수보다 큰지를 확인하는... 대체로 이 온전한 상태의 채크의 목적은 훼손된것으로 의심이 되는 슈퍼블럭을 찾는것이다. 이 경우에 시스템은 슈퍼블럭의 대안의 복사본을 사용하도록 결정한다.
- Free Blocks
다음으로 fsck 는 파일시스템의 구조를 만들기 위해 파일 시스템 내에 현재 할당되어 있는 indoe, indirect block, double indirect block 등등을 스캔한다. 그것은 할당된 비트맵의 올바른 버전을 생산하기 위한 정보를 사용한다. 따라서, bitmap 과 inode 사이에 inconsistenct 가 있다면, indoe 안의 정보를 신뢰함으로써 조치된다. 이러한 채크는 모든 inode 에 실행된다. 사용되고 있는 것으로 보이는 모든 inode 가 inode bitmap 들 같은곳에 마크 되어 있는지를 확인하기 위해서이다.

1. Inode 상태각각의 idnoe 는 훼손이나 다른 문제가 채크 된다. 예를들어 fsck 는 각각 할당된 inode 가 유효한 타입의 필드를 가지고 있는지 확인한다.(예를 들어 정규파일, 디렉토리, 심볼릭 링크등) 만약 아이노드 필드에 문제가 있다면 고치가 쉽지 않다. inode 는 의심되고 fsck 에 의해 지워진다. inode bitmap 은 이에 상응하여 수정된다

2. Inode linksfsck 는 역시 각각 할당된 inode 의 link count 도 확인한다. 당신이 recall 을 할때, link count 는 이 특별한 파일을 위한 reference (즉, 링크)를 담고 있는 다른 디랙토리들의 숫자를 가르킨다. 이 link count 를 확인하기 위해서, fsck 는 전체 디렉토리 트리를 통해서 스캔한다. 루트 디렉토리에서 시작하고, file system 안의 모든 파일과 디렉토리를 위한 자신의 link count 를 계산한다. 새롭게 계산된 count 와 미스매치가 있으면, 교정이 반드시 취해진다. 보통은 inode 안의 값을 수정한다. 할당된 indoe 가 발견되지만 어떤 디랙토리도 이를 참조하지 않으면 이것은 lost+found 디렉토리로 옮겨진다.

3. Duplicate

fsck 는 중복 포인터(즉, 두개의 inode 가 같은 block 을 가르킴)도 check 한다. 하나의 inode 가 명백하게 잘못되었다면, 그것은 삭제된다. 그렇지 않으면 대안적으로 가르키던 block 은 복사된다. 따라서 각각의 inode 는 원하는데로 자신의 복사본을 가지게 된다.

4. Bad block

배드블럭 포인터의 책크는 모든 포인터 리스트를 스캐닝 하는동안 수행된다. 만약 그것의 유효한 범위를 넘어서는 무언가를 명백하게 가르키고 있다면 그것은 "bad"로 간주된다. 예를 들어, 파티션 사이즈보다 더 큰 블럭을 참조하는 어드래스를 가지고 있는 경우이다. 이 경우에 fsck 는 지능적인 어떠한 것을 할 수 없다. 이것은 단지 indoe 나 indirect block 의 포인터를 지운다.

5. Directory check

fask 는 유저파일의 컨텐츠를 이해하지 못한다. 그러나 디렉토리는 특정한 포맷정보를 담고 있다. 따라서 fsck 는 각각의 디렉토리의 컨텐츠에 대해서 부가적인 온전성 채크를 한다. 다음을 확인한다.
- "." 과 ".."이 첫번째 엔트리들인지.
- 각각 아이노드가 할당된 디렉토리 엔트리를 참조하는지,
- 엔트리 하이락키안에 한번 이상 링크된 디렉토리가 없는지 (엔트리 트리에서 중복되어 링크된 디렉토리가 없는지...)

보았듯이, fsck 의 동작을 완성하는것은 파일시스템의 복잡한 지식이 필요하다. 모든 경우에서 정확하게 동작하는 코드를 확인하는 것은 도전적일 수 있다. 그러나 fsck (를 비롯한 유사한 것들)은 크고 어쩌면 근본적인 문제를 가지고 있다. 그들은 너무 느리다. 커다란 디스크 볼륨에서, 모든 항당된 블럭을 찾고, 각 디렉토리 트리를 읽고 확인하기 위해 모든 이스크를 스켄하는 것은 수분에서 수시간이 걸린다. 디스크의 용량이 커지고, RAID들이 인기가 높아짐에 따라 fsck 의 성능은 엄두도 못낼정도로 비싸졌다.
높은 레벨에서, fsck 의 기본적인 전제는 조금 비이성적이 되어보인다. 단지 새블럭의 업데이트 동안 발생한 문제를 해결하기 위해 온 디스크를 스캔하는 것은 믿을 수 없을 정도로 비싸다. 이상황은 너의 침대 바닦에 키를 떨어뜨리고 온 집안들 찾는 것과 같다.

42.3 Solution #2: Journaling (orWrite-Ahead Logging) Probably

아마도, 업데이트 consistent 문제의 가장 유명한 해결책은 dbms 의 세계로부터 아이디어를 훔쳐왔다. write-ahead logging 으로 알려져 잇는 이 아이디어는 정확하게 이러한 종류의 문제를 고심해서 발명되었다. 파일시스템에서, 우리는 역사적인 이유로 write-ahead logging 을 journaling 이라고 부른다. 이것을 수행한 첫 파일시스템은 Cedar 이다. 많은 현대적은 파일시스템이 이 아이디어를 사용한다.Linux ext3 and ext4, reiserfs, IBM's JFS, SGI's XFS, andWindowsNTFS.

기본아이디어는 다음과 같다. 디스크에 업데이트 할때, data structure 들이 맞는 장소에 쓰여지기 전에, 먼저 작은 노트 (디스크에 어딘가 다른 , 잘 알려진 곳)에 네가 무엇을 할 것인지를 쓴다. 이 노트를 쓰는것이 "write ahead" part 이다. 그리고 우리는 그것을 structure(우리는 "log"를 조직화한다) 에 쓴다. 이러한 이유로 write-ahead logging 이다.

디스크에 note 를 쓰는것으로 인해, 우리는 업데이트 하는 structure들의 update 동안 crash 가 발생한다면, 노트를 보고 예전으로 돌아간 이후에 다시 시도 할 수 있는것을 보장할 수 있다.따라서, 너는 크래쉬 후에 무엇을 어떻게 수정해야 하는지 확실히 알 수 있다. 모든 디스크를 스캔하는 것 대신에.. 디자인적으로, 따라서 디자인 적으로 저널링은 업데이트하는 동안에 약간의 일이 더해진다. 리커버리 동안의 요구되는 일의 양을 엄청나게 줄이기 위해서..

우리는 유명한 파일시스템인 linux ext3 를 묘사해 볼 것이다. 이것은 파일시스템에 journaling 을 포함하고 있다. 디스크 structures 의 대부분은 ex2 와 동일하다. 디스크를 block group 들로 나뉘어 지고, 각 block group 은 inode bitmap 와 data bitmap 들을 가지고 있고 더하여, indoe 들과 data block 들을 가지고 있다.(주** inode bitmap, data bitmap, inode, data block 을 각 block group 들이 가지고 있다.) 새로운 key structure 는 저널 그 자체 이다. 저널은 파티션이나 다른 디바이스장치안에 작은양의 공간을 차지하고 있다. 따라서 ext2 file system (저널이 없는)은 아래와 같다.

같은 파일시스템 이미지 안에 저널이 있다고 가정하자. (비록 때때로 분리된 디바이스에 있거나 파일시스템 안에 파일로 있기도 하지만..) ext3 file system 은 아래와 같다.

Data Journaling

data journaling 이 어떻게 동작하는지를 이해하기 위해서 단순한 예제를 보자. Data journaling 은 이 논의의 기초의 많은 부분을 차지하는 ext3 파일 시스템에 사용가능한 모드이다.

5개의 블럭을 쓴다. - Txb : Transaction begin
update 에 대한 정보
I[v2], B[v2]와 Db 의 마지막 주소등
TID(transaction identifier)
- I, B, D blocks

이것이 physical logging 으로 알려진, 우리가 저널에 업데이트 하는 정확한 물리 컨텐츠이다. (대안의 아이디어로 logical logging, 은 좀더 소형의 logical 묘사를 저장한다.. 예를 들면, 이 업데이트는 파일 X에 데이터 블럭 Db 를 확장하려고 한다. 이는 좀더 복잡하지만, 로그의 공간을 줄이고 아마도 퍼포먼스를 증대시킨다.)

- TxE : trahsaction end
TID

이 트렌젝션은 디스크를 안전하게 하기 때문에, 우리는 파일시스템의 오래된 structure에 overwrite 할 준비가 되어 있다. (주** structure 는 이 문서에서 journal 이 아닌 파일시스템 상에 metadata 나 data 가 저장되는 공간이나 위치등을 뜻한다.)
이 프로세스에서 이를 checkpointing 이라고 한다.
따라서, 파일시스템 checkpoint 를 위해서 (즉, 저널 안에 미결된 업데이트의 데이터를 처리하는것), 우리는 I[v2], B[v2] , Db 의 write 를 시도한다. 그들의 디스크 위치에.
만약 우 쓰기가 성공적으로 완료되면, 우리는 파일시스템에 성공적인 check point 를 수행한 것이다. 그리고 기본적으로 완료된다.

따라서, 동작의 초기 시퀀스를 아래와 같다.1. journal write
TxB를 포함한 transaction 을 로그에 쓰고, 모든 pending data 와 metadata 를 로그에 업데이트 하고, TxE 를 로그에 쓴다. 이 쓰기가 완료되는것을 기다린다.
2. checkpoint
미루어진 metadata 와 data update 들을 filesystem 의 그들의 최종위치에 write 한다.

이 예에서 저널에 먼저 다음을 쓴다.TxB, I[v2], B[v2], Db, TxE

이 쓰기가 완료된 이후에, checkpointing 을 완료 한다.
I[v2], B[v2], Db 를 디스크에 최종 위치에 쓴다.

저널에 쓰기가 진행되는 동안 crash 가 발생하였을대 작은 어려움이 있다. 여기서, 우리는 트랜젝션(예를 들면, TxB, I[v2], B[v2], Db, TxE) 안에 블럭의 셋을 디스크에 쓰려고 노력한다. 이것을 하는 단순한 방법은 한번에 하나씩 시도하고 쓰기가 완료되는 것을 기다린후 다음것을 시도하는 것이다. 그러나 이것은 느리다. 이상적으로는, 우리는 5블럭을 한번에 쓰기를 원한다. 이것은 single sequential write 안에 5개의 write 들이 들어가고 따라서 더 빠르다. 그러나 이것은 안전하지 않다. 다음과 같은 이유이다. : big write 가 주어진 상황에서, disk 는 내부적으로 스케줄링을 수행할것이다. 작은 주문안의 큰 write 의 작은 조각을 완료한다. 따라서 디스크는 내부적으로 (1) TxB, I[v2], B[v2], TxE 를 쓰고
(2) 나중에 Db 를 쓸것이다.
운나쁘게, 디스크가 (1) 과 (2) 사이에 파워를 잃으면 디스크는 무엇에 처하게 될까.

무엇이 문제인지 보자.
이 트렌젝션은 유효한 트렌젝션처럼 보인다.(그것은 매칭되는 시퀀스 넘버(TDI)를 가진 TxB 와 TxE 를 가지고 있다.) 게다가 파일 시스템은 4번째 블락(Db) 를 알아볼 수 없고 그것이 잘못되었는지 알수 없다. (결국에는 그것이 임의적인 유저데이터 일수 있다.) 따라서 파일시스템이 지금 리붓한다면, 그리고 리커버리를 실행한다면, 그것은 이 트렌젝션을 재 실행할 것이다. 그리고 깨어진 블럭 '??' 의 담긴 내용을 Db 가 존제하는 위치로 부터 복사하는것을 무시할 것이다. 이것은 file 안의 임의적인 유저데이터에게 좋지 않고, 파일 시스템의 치명적인 부분에는 더 좋지 않다. superblock 이 망가진 경우 파일시스템 마운트가 불가능한 상황을 만든다.

따라서, 우리의 현재의 파일시스템의 업데이트 프로토콜은 3가지 단락으로 라벨링 된다.
1. journal write
TxB, metadata, data 를 포함하는 transaction 의 contexts 들을 log 에 쓴다. 이것이 완료되는것을 기다린다.

2. journal commit
transaction commit block(TxE 를 포함하는) 을 log 에 쓴고, 완료되는것을 기다린다. 이후에 transaction 은 committed 되었다고 한다.

3. checkpoint
update 된 metadata 와 data의 의 contect들을 disk 의 최종 위치에 write 한다.

Recovery
Recovery
파일 시스템이 크래쉬로부터 어떻게 저널을 사용해서 리커버리를 수행하는지 보자.

= 크래쉬가 TxE 가 쓰기 전에 발생하면, 일은 쉽다. pending update (file structure 에) 를 단순히 skip 한다.

(주** pending update 는 주로 실제 디스크에 최종 위치에 쓰는것을 뜻함)

= 크래쉬가 commit 뒤에 checkpoint 전에 났을 경우.
시스템이 부팅할때, 파일 시스템 리커버리 프로세스가 로그를 스캔하고 디스크에서 커밋된 트랜젝션을 살펴본다. 이 트랜젝션은 순서대로 replay 된다. 그리고 이 블럭들은 그들의 최종 디스크 위치에 write 를 시도한다. 이 로깅 방식은 가장 단순한 방

식중에 하나이고, redo logging 이라고 부른다.

= 크래쉬가 checkpointing 중 어느 시기에 발생하더라도 괜찮다는것에 주목해라. 심지어 블럭중 몇몇이 최종 위치에 업데이트가 된 이후에라도.
최악의 상황에, 이 업데이트중 몇몇은 리커버리 동안에 다시 수행된다. 리커버리는 가끔 일어나는 동작이기 때문에 몇가지 불필요한 wrtie 들은 큰 걱정이 아니다.

Batching Log Updates
(journal log commit 의 일괄처리)

당신은 많은 부가적인 디스크 트래픽을 더할 수 있는 기본 프로토콜에 대해 주목했을 것이다. 예를 들어 file1 과 file2 라는 두개의 파일을 같은 디랙토리에 연이어 생성한다고 상상해 보자. 하나를 생성하기 위해, 몇게의 최소한의 on-disk structure들을 어베이트 해야만 한다.(최소한 새로운 inode 를 할당하기 위해서 inode bitmap, 파일의 새로 생성되는 아이노드, 새로운 디랙토리 엔트리를 담기위해 부모 디랙토리 데이터 블럭의 수정, 더나아가 부모 디렉토리의 idnoe)
저널 모드에서, 논리적으로 우리는 이 모든 정보들을 저널에 commit 해야 한다. 두파일의 생성시 각각
이파일은 동일한 디렉토리에 있는 파일이기 때문에 같은 inode block 안에 inode 들을 가진다고 추정해보자. 이것은 우리가 조심하지 않으면, 우리가 같은 블럭에 반복해서 write 를 하는것에 처하게 된다는 것을 의미한다.

이문제의 해결을 위해서, 몇몇 파일 시스템은 매번 디스크 업데이트 때마다 commit 을 하지 않는다. (예를 들면 Linux ext3 같은 경우..) 더 정확히 말하면, 모든 전역적인 트렌젝션을 하나에다 버퍼링 한다. 위에 있는 우리의 예에서, 두개의 파일이 생성될때, 파일 시스템은 단지 dirty 가 된 in-memory 안의 inode bitmap, file 의 inode들, 디렉토리 데이터, 디렉토리 inode 을 mark 만 한다. 그리고 그들을 현재 트렌젝션으로부터의 블럭 리스트에 추가한다. 마침내 이 블럭들이 디스크에 쓰여질때, (말하자면 5초 타임아웃 후에) 이 하나의 단일 전역 트렌젝션은 위에 묘사한 모든 업데이트를 포함하여 커밋된다. 따라서 버퍼링 업데이트에 의해서, 파일 시스템은 많은 경우에 디스트 쓰기 트래픽이 생기는 것을 피할 수 있다.

Making The Log Finite
(로그의 제한 만들기)

파일시스템 버퍼는 때때로 memory 안에 update 한다. disk 에 쓰기위한 최종 시간이 되었을때, 파일 시스템은 첫번째로 (write-ahead log라 부르는) journal 에 트렌젝션의 디테일을 조심스럽게 쓴다. 이후에 트렌젝션이 완료 되었을때, 파일 시스템은 디스크의 그들의 최종 위치로 그 블럭들을 checkpoint 한다.

그러나, journal log 는 한정된 사이즈 이다. 만약 우리가 아래 그림처럼 트랜젝션을 더하면 그것은 바로 가득찰 것이다. 그 이후에 무엇이 일어날것이라고 생각하는가?

로그가 차기 시작할때 두가지 문제가 발생한다. 첫째는 단순하고 덜 위험하다. 로그가 커질수록 리커버리 할 것이 커진다. 리커버리 프로세스는 리커버를 위해 로그안에 트렌젝션들을 순서대로 replay 해야 할 것이다.

두번째는 좀더 이슈이다.
로그가 가득차거나 거의 가득 찼을때, 디스크에 더이상 트랜젝션들이 commit 될 수 없다. 따라서 파일시스템이 "less than

useful" (즉, 쓸모없는) 상태가 된다.

이 문제들을 고심해서 다루기 위해, 저널 파일 시스템은 로그를 원형 자료구조 처럼 다룬다. 이것이 저널이 대때로 환형로그 처럼 나타내어 지는 이유이다. 그렇게 하기 위해서 파일 시스템은 checkpoint 이후에 몇번의 행동을 취해야 한다. 특히, 하나의 트렌젝션이 checkpoint 되고, 파일 시스템은 저널안에 차지하던 공간을 free 해 주어야만 한다. 로그 공간을 재사용하기 위해서... 이 끝을 달성하는 많은 방법이 있다. 예를 들면 journal superblock 안에 이 로그안에서 가장 오래된 트렌젝션과 가장 최근의 트랜젝션을 마킹한다. 다른 영역들은 자유롭게 사용하면 된다. 여기 이 매커니즘의 묘사가 그림으로 있다.

저널 슈퍼블럭 안에 (main file system superblock 과 혼동하지 말자), 저널 시스템이 충분한 정보를 저장한다. 어느 트렌젝션이 아직 checkpoint 되지 않았는지 알기 위해, 따라서 리커버리 타임이 절감된다. 더 나아가 환형 방식에서 로그의 재사용이 가능하다.

그리고 우리의 기본 프로토콜에 몇 스탭을 추가한다.
1. Journal write
...
2. Journal commit
...
3. checkpoint
...
4. free
얼마후에, journal superblock 의 갱신에 의해 journal 안에 transaction free 가 mark 된다.

< Metadata Journaling >

Metadata Journaling
리커버리가 이제 빠르기는 하지만(전체 디스트를 스캔하는것에 견주어서 저널을 스켄하고 몇개의 트렌젝션을 replay 한다.) 파일시스템의 통상동작은 우리가 바라는것보다 느려졌다. 특히, 디스크에 쓰기위해서, 우리는 저널을 먼저 기다리조 있다. 따라서 이중 write 프래칙이다. 이 더블링은 특히 연속되는 쓰기 워크로드 동안에 가중된다. 이 워크로드는 그 드라이브의 write 밴드위스의 피크의 절반을 진행할 것이다. 게다가 저널의 쓰기와 메인 파일 시스템에 쓰기사이에는 비싼값을 치루는 seek 가 있다. 몇몇 워크로드를에 명백한 오더해드를 더하는...

사람들이 퍼포먼스의 속도향상을 위해서 약간씩 다른 것들을 시도하는것은, 디스크의 모든 데이터 블럭의 두번쓰는 높은 비용 때문이다. 예를들어, 우리가 묘사하는 저널링 모드는 종종 data journaling 이라고 불리운다. 저널링의 단순하고 좀더 흔한 방식은 때때로 ordered journaling 이나 metadata journaling 이라고 불리운다. 그리고 그것은 거의 같다. user data가 journal 에 저장되지 않는 것만 제외하고, 따라서, 위의것과 같은 업데이트를 수행할때 저널에는 다음과 같이 쓰여진다.

앞에선 로그에 써졌던, 데이터 블럭 Db는 추가 write 를 피해서 파일시스템에 적절하게 써진다. 디스크의 대부분의 io 트레픽은 데이터였다. 데이터를 두번쓰지 않는 것은 저널의 IO 로드를 상당히 줄여준다.그렇긴 하지만, 우리에게 이 수정은 언제 데이터 블럭을 디스크에 쓰는것이 좋을까라는 흥미로운 질문을 던진다.

밝혀진것처럼, 데이터 쓰기의 순서는 메타데이터 저널링에서 매우 중요하다. 예를 들면, 우리가 I[v2] 와 B[v2]를 포함하는 트렌젝션이후에 디스크에 Db 를 쓴다면 어떨까? 운나쁘게 이접근은 문제가 있다. I[v2]와 B[v2] 가 쓰여졌지만 Db 는 디스크에

없는경우를 고려해보자. 파일시스템은 리커버를 시도할 것이다. 왜냐하면 Db 가 로그에 없기 때문이다. file system 은 I[v2]와 B[v2]를 replay 하려고 할것이고 (파일시스템 메타데이터관점에서) file system 의 consistent 가 보장되게 된다. (Db는 써지지 않았는데도 불구하고..)그러나 I[v2]는 쓰래기 데이터를 가리키고 있다. 즉, Db 가 있어야할 슬롯에 무엇이 있는지 모른다. 이러한 상황이 발생하지 않기 위해서, 몇몇 파일 시스템은 (ext3 같은) data block write 을 디스크에 먼저 쓴다. 메타데이터를 쓰기 전에.

이 프로토콜은 다음을 따른다.
1. Data write
최종 파일시스템 위치에 data(Db) 를 filesystem structure 에 write 하고 완료를 기다린다. (대기는 옵션이다.)
2. Journal metadata write
TxB 와 I[v2], B[v2] (meta data) 를 journal log 에 쓰고, 완료되기를 기다린다.
3. journal commit
TxE 를 포함하는 transaction commit block 을 journal log 에 쓰고, 완료를 대기한다. Transaction (data 를 포함한) 은 이제 commit 되었다.
4. checkpoint metadata
I[v2], B[v2](metadata) 를 파읽스템 안에 그들의 최종 위치에 write 한다.
5. Free
후에, journal superblock 안의 transaction free 가 mark 된다.

정말로 "포인트되는 object 를 그것을 가르기는 object 전에 써라" 라는 룰이 crash consistency 의 핵심룰이다.

ext3 의 ordered journaling 과 유사한 metadata journaling 은 full data journaling 보다 더 인기가 있다. 예를 들어 윈도우의 NTFS, SGI의 XFS 는 둘다 non-ordered metadata journaling 을 사용한다.

최종적으로 (위의) journal 에 쓰기가 issue 되기 전에 step1 인 data 쓰기가 완료되는것을 강제하는것에 주목해 보자.
(step2 - metadata journal write) 는 위의 프로토콜이 지시하는 것처럼 정확함을 요구하지 않는다. 특히, data write 뿐만아니라, TxB 와 metadata 가 issue 되는것이 괜찮다.

진짜 요구사항은 step 1과 2 가 journal commit block (step 3) 의 issue 이전에 완료되는 것이다.

Tricky Case: Block Reuse
까다로운 경우 : 블럭의 재사용

저널을 만드는것을 좀 더 까다롭게 하는, 흥미있는 케이스가 있고, 논의할 가치가 있다. 그들의 다수는 블럭 재사용을 하며 순환된다. Stephen Tweedie 는 다음과 같이 말했다."모든 시스템에서 끔찍한 부분은 무었인가? 그것은 파일 삭제이다. 삭제의 모든 부분이 아슬아슬 하다. 삭제의 모든 부분이... 블럭이 삭제되고, 재사용 될때 무슨일이 일어나는지는 악몽을 꾸는것이다."

메타데이타의 저널링의 어떤 방법을 사용한다고 가정하자. 파일을 위한 데이터블록은 저널되지 않는다. (주** metadata journaling mode 이다.) 네가 foo라고 불리는 디렉토리를 가지고 있다고 해 보자. 유저는 foo 에 엔트리를 추가한다. (파일을 생성하는것을 말하는 것이다.) 그리고 (디렉토리는 메타데이터로 고려되므로) foo 의 내용물은 log 에 써진다. foo 디렉토리 데이터의 위치가 block 1000 이라고 해보자. log(journal)은 다음과 같이 되어있다.

이상황에서 사용자는 디렉토리 안의 모든것과 디렉토리를 삭제한다. 재사용을 위해 block 1000 은 free된다. 마지막으로 사용

자는 새로운 파일을 만든다. (foobar 라고 하자.), 이때, foo 가 사용하던 같은 블럭 1000 을 재사용한다. foobar 의 inode 는 disk 에 commit 된다. 그것의 data도 그렇다(주** 잘못 쓴듯.) 그러나 metadata journaling 이 사용되기 때문에, 저널에는 foobar 의 inode 만이 commit 된다. foobar 파일안의 block 1000안데 새롭게 쓰여진 뎅터는 저널되지 않는다.

이제 이 정보들이 로그안에 있는 상태에서 크래쉬가 발생했다고 해보자. 리플레이 동안에 리커버리는 로그안에 모든것을 replay 한다. 이 replay 는 foobar 파일의 data 를 오래된 directory contects로 overwrite 하게 된다. 명백하게 이것은 올바른 리커버리가 아니고, 확실히 foobar 의 파일을 읽었을때 놀라게 될 것이다.

Wrapping Up Journaling: A Timeline

(주** : 아랫방향으로 시간이 증가한다.)(주** : 점선으로 나뉘어진 부분사이의 order 는 지켜져야 한다. 점선 안의 transection 의 order 는 상관 없다.)

42.4 Solution #3: Other Approaches

42.4 Solution #3: Other Approaches

우리는 파일시스템 메타데이다의 consistent 를 지키는 두 옵션에 대해서 길게 이야기 하였다.

- fsck 를 이용한 lazy 접근
- journaling 을 이용한 보다 적극적인 접근

그러나 오직 두가지 접근만 있는것은 아니다.

- Soft Updat

접근중 하나는 "Soft Update" 로 알려져 있고 Ganget and Patt 에 의해 소개되었다. 이 접근은 파일시스템에 모든 write 를 조심스럽게 ordering 한다. 디스크 구조가 절대 inconsistent 상태로 남아있지 않도록 확인하기 위해서 이다. 예를 들면, 포인트가 되는 데이터 블럭은 그것을 포인트 하는 inode 보다 먼저 쓰는 방법으로, 우리는 inode 가 절대 쓰래기값을 포인트 하지 않는것을 확인할 수 있다.; 유사한 방법으로 파일시스템의 모든 구조를 운용할 수 있다. Soft Update들의 구현은 도전적이다. 그러나 위에 묘사한 저널링 레이어는 비교적 정확환 파일 시스템 구조의 작은 이해만으로도 구현할 수 있다. Soft Update는 각각의 파일시스템 데이터 구조에 대한 복작한 지식을 요구하고, 시스템에 그정도의 복잡성을 더한다.

- copy-on-write(COW)

또다른 접근은 copy-on-write(COW)로 알려져 있고, 많은 유명한 파일시스템이 사용하고 있다. Sun 의 ZFS 를 포함해서.. 이 기술은 제자리에 있는 파일이나 디렉토리를 절대 overwrite하지 않는다. 더 정확히 말하면, 새로운 업데이트를 디스크에 사전에 준비된 사용되지 않은 영역에 업데이트 한다. 많은 업데이트들이 완료된 이후에, COW 파일 시스템은 새롭게 업데이트된 structure들을 가르키는 포인터가 포함된 파일 시스템의 root structure들을 뒤집는다. 이 실행은 복잡하지 않은 파일 시스템의 consistent 를 유지하도록 만든다. 우리가 다른 챕터에서 LFS(log-structured file system)에 대해 논의할때 이 기술에 대해서 좀 더 많이 배울 것이다.

- BBC(backpointer-based consistency)

다른 접근은 Wisconsin 에서 개발된 것이다. 이 기술은 BBC(backpointer-based consistency)라고 이름지어졌다. write 사이에 ordering 은 강요되지 않는다. consistency 를 만족하기 위해서, 부가적인 back pointer(주** 예를 들면 data 자신을 가르키 indoe 가 있다면 data 의 back point 는 이 inode 를 가르킨다)는 가 시스템의 모든 block 에 추가된다. 예를 들면, 각각의 데이터 블럭은 그것이 속하는 inode 를 위한 참조를 가진다. 파일을 엑세스할때, 파일 시스템은 그 파일이 checking에 의해서 consistent 한지 결정할 수 있다. 그 checking 은 앞의 포인터(예를 들면, inode 나 direct block 안의 주소값)이 그것을 위한 백을 참조하는 블럭을 가르키는지 아닌지를 본다. 만약 그렇다면, 모든것은 안전하게 이스크에 도달하고 파일은 consistent 하

다. 만약 그렇지 않다면, 파일은 inconsistent 하고, error 가 리턴된다. back pointer 를 파일시스템에 추가함으로써, lazy crash consistency의 새로운 방법이 이루어 질 수 있다.

마지막으로, 우리는 디스크 쓰기가 완료되는것을 위해 기다리는 journal protocol 의 대부분의 시간을 줄일 수 있는 기술에 대해 탐험해본다. 이는 optimistic crash consistency 가고 한다. 이 새로운 접근은 디스크에 가능한한 많은 write 가 issue 되고, transaction checksum 의 일반적인 방법을 사용한다. 더나아가 약간의 다른 (그들이 생성시킬 수 있는 inconsistency들을 탐지할 수 있는)기술을 사용한다. 몇가지 workload들을 위해서, 이 optimistic 기술은 중요도의 order를 통해서 성능을 향상시킬 수 있다. 그러나 정말 잘 동작하기 위해서, 조금 다른 디스크 인터페이스가 필요된다.