45

크래시 일관성 : FSCK와 저널링

이제까지 본 바와 같이, 파일 시스템은 그 기본 개념들을 구현하는 데 필요한 각종 자료 구조들을 관리한다. 이 자료 구조에는 파일, 디렉터리, 그 외에 각종 메타데이터들이 있다. 여타 자료 구조와는 다르게(예를 들어 실행 중인 프로그램들이 사용하는 메모리 상의 자료 구조) 파일 시스템의 자료 구조는 **안전**하게 저장되어야 한다. 즉, 장시간 사용 후에도 유지되어야 하며 전력 손실에도 하드 디스크나 플래시 기반 SSD 장치의 데이터는 손상없이 유지되어야 한다.

파일 시스템이 가진 가장 큰 어려움은 전력 손실이나 시스템 크래시가 발생하는 상황에서도, 어떻게 안전하게 디스크 상의 내용을 갱신하는가에 대한 문제이다. 만약 디스크 자료 구조를 갱신하는 도중에 누군가가 전원 선에 걸려 넘어져서 기계가 꺼졌다면 어떻게 되는가? 운영체제가 버그 때문에 멈춘 경우는 어떻게 하는가? **전력 손실**이나 **크래시** 때문에 디스크 상의 자료 구조를 안전하게 갱신하는 것은 상당히 까다로운 작업이 된다. 파일 시스템은 **크래시 일관성**(**crash-consistency**)이라는 새롭고 흥미로운 문제에 직면하게 된다.

문제를 이해하는 것은 어렵지 않다. 어떤 특정 작업을 위해 디스크 상에서 두 개의 자료 구조 A와 B를 갱신해야 한다고 해 보자. 디스크는 한 번에 하나의 요청만 처리할 수 있기 때문에 두 요청 중 하나의 요청이 먼저 디스크에 도달할 것이다(A 또는 B). 하나의 쓰기 작업만 완료한 상태에서 시스템 전원이 나간 경우, 디스크 상의 자료 구조는 일관성이 깨지게(**inconsistent**) 된다. 이러한 특성 때문에, 파일 시스템에서는 이제까지는 없었던 새로운 문제를 직면한다. 크래시 일관성 문제이다.

핵심 질문 : 크래시에도 불구하고 디스크 갱신하기

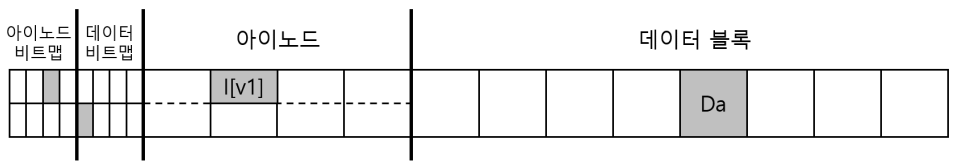
두 쓰기 동작 사이에 시스템은 크래시되거나 전력이 끊어질 수도 있기 때문에 디스크 상의 상태는 부분적으로만 갱신이 될 수도 있다. 크래시 이후에 시스템이 재구동되면 파일 시스템을 다시 마운트하려고 할 것이다(파일 접근 등의 동작을 위해서). 임의의 시간에 크래시가 발생할 수 있다고 하면 파일 시스템이 어떻게 해야 디스크 상의 자료를 올바른 상태로 유지할 수 있을까?

이번 장에서는 이 문제에 대해서 좀 더 자세히 다룰 것이다. 이에 대한 해결책들을 살펴보기로 한다. 먼저 과거 파일 시스템들이 사용했던 **fsck** 또는 **파일 시스템 검사기**라는 방법을 살펴보도록 하겠다. 그 다음, **저널링**(**journaling**) 또는 **write-ahead logging**(WAL)이라는 다른 기법을 살펴보겠다. 저널링이나 WAL 기법은 쓰기 오버헤드가 추가되기는 하지만 전력 손실이나 크래시 상황으로부터 좀 더 빠르게 복구할 수 있다. Linux ext3가 구현하고 있는 몇 가지 기본적인 저널링 기법들을 살펴볼 것이다 [Twe98; PAA05].

45.1 예제

저널링을 이해하기 위해 예제 하나를 살펴보자. 디스크 상에서 여러 개의 자료 구조를 갱신하는 연산을 예로 들겠다. 워크로드는 기존 파일에 블럭을 하나를 추가하는 연산이다. 파일을 열고 `lseek()`로 파일의 끝으로 오프셋을 이동한 후에 4 KB 를 쓰고, 파일을 닫는다.

이전에 사용했던 간단한 파일 시스템 자료 구조를 가정한다. 이 예제에서는 **아이노드 비트맵**(아이노드당 하나씩, 8비트 크기), **데이터 비트맵**(마찬가지로 데이터 블럭당 하나씩, 8비트 크기), 아이노드(총 8개, 0부터 7의 번호를 갖고 4개의 블럭을 사용) 그리고 데이터 블럭(총 8개, 0부터 7의 번호를 가짐)이 존재한다. 파일 시스템의 구조는 다음과 같다.



아이노드 테이블을 보면 2번 아이노드가 할당되어 있다. 아이노드 비트맵에 세 번째 비트가 설정되어있다(아이노드 번호 2). 이 파일은 4번째 데이터 블럭을 사용하고 있다. 데이터비트맵의 4번째 비트가 사용 중으로 표시되어 있다. 아이노드는 첫 번째 버전이기 때문에 I[v1]로 표기되어 있다. 곧 변경될 것이다(앞에서 설명한 워크로드의 동작으로 인해서).

아이노드의 구조를 살펴보도록 하자. I[v1] 내부는 다음과 같다.

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

파일의 크기는 `1`이고(한 블럭이 할당됨) 첫 번째 직접 포인터는 `4`번 블럭을 가리키며(파일의 첫 번째 데이터 블럭은 Da), 나머지 직접 포인터들은 `null`로 설정되었다

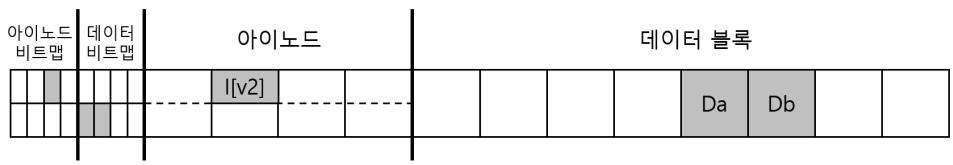
(사용하고 있지 않음을 표시함). 물론, 실제 아이노드는 더 많은 항목들이 있지만 더 자세한 정보는 이전 장을 참고하기 바란다.

이 파일의 끝에 내용을 추가한다는 것은 새로운 데이터 블럭을 추가하는 것이다. 세 개의 디스크 자료 구조를 갱신해야 한다. 아이노드(새로운 데이터 블럭을 가리켜야 할 뿐만 아니라 추가로 인해 더 커진 크기를 반영하기 위해)와 새로운 데이터 블럭 Db, 그리고 데이터 비트맵(B[v2]라고 하자)이다. 새로운 블럭은 사용 중이 된다.

메모리에는 디스크에 기록할 세 개의 블럭이 존재한다. 갱신된 아이노드(아이노드 버전 2 또는 짧게 I[v2])는 다음과 같다.

```
owner          : remzi
permissions    : read-write
size           : 2
pointer        : 4
pointer        : 5
pointer        : null
pointer        : null
```

데이터 비트맵(B[v2])은 00001100로 갱신되었다. 마지막으로 사용자의 내용이 저장된 데이터 블럭(Db)이 있다. 이들이 디스크에 성공적으로 기록되면, 디스크의 모습은 아래와 같다.



위와 같은 형태가 되기 위해서 파일 시스템은 디스크에 세 번의 쓰기를 수행해야 한다. 하나는 아이노드(I[v2]), 그 다음은 비트맵(B[v2]), 그리고 데이터 블럭(Db) 쓰기이다. `write()`의 결과는 디스크에 즉시 반영되지 않는다. 대신 변경된 아이노드와 비트맵 그리고 새로운 데이터는 일정 기간 동안 메인 메모리 상에 존재하다가(**페이지 캐시**나 **버퍼 캐시**에) 파일 시스템이 실제로 디스크를 실행할 때(5초나 30초 이후 정도에) 기록된다. 하지만 크래시 발생으로 인해서 디스크 기록 과정이 엉망이 될 수가 있다. 구체적으로 살펴보기 위해 세 개의 쓰기 중에 하나나 두 개의 쓰기만이 실제로 있었다고 해 보자. 파일 시스템은 이상한 상태에 놓이게 될 것이다.

크래시 시나리오

이 문제를 더 잘 이해하기 위해서 크래시 시나리오를 예로 들어보자. 한 번의 쓰기만 성공한 경우를 생각해 보자. 다음 세 가지 경우가 있을 수 있다.

- **데이터 블록(Db)만 디스크에 기록됨**. 이 경우에 데이터는 디스크에 있지만 그것을 가리키고 있는 아이노드가 없으며 할당 여부를 나타내는 비트맵도 없다. 이러한 경우는 파일 시스템의 크래시 일관성 측면에서는 문제가 없다[1].

1) 하지만 데이터의 일부를 잃었기 때문에 사용자에게는 문제가 될 수가 있다.

- **갱신된 아이노드(I[v2])만 디스크에 기록됨**. 이 경우에는 Db가 기록되려던 디스크의 주소(5)를 아이노드가 가리키고 있지만 아직 Db는 그 자리에 기록 되지 않았다. 그러므로, 포인터를 그대로 읽는다면 디스크에서 의미없는 데이터를 얻게 된다(기존의 디스크의 주소(5)에 있던 데이터).
  더 나아가 파일 시스템의 "**일관성 손상**"이라는 새로운 문제를 만난다. 디스크 상의 비트맵은 데이터 블록 5번이 할당되어 있지 않다고 하지만 아이노드는 할당되어 있다고 한다. 파일 시스템의 자료 구조들 간의 이러한 불일치로 파일 시스템 전체의 일관성이 손상되게 된다. 파일 시스템을 사용하기 위해서는 이러한 문제를 어떤 식으로든 해결해야 한다(이제 다룰 것이다).
- **갱신된 비트맵(B[v2])만 디스크에 기록됨**. 이 경우의 비트맵은 블록 5번이 할당되었다고 표시하지만 아이노드가 가리키고 있는 블록은 없다. 그러므로 파일 시스템의 일관성이 손상된 상태가 되었다. 해결하지 않고 그대로 둔다면 블록 5번은 파일 시스템에서 사용될 수 없다.

세 개의 블록을 디스크에 기록하는 시나리오에서 세 개의 또 다른 크래시 시나리오가 있을 수 있다. 두 개의 쓰기가 성공하고 마지막 한 개의 쓰기는 실패한 경우이다.

- **데이터(Db)를 제외한 아이노드(I[v2])와 비트맵(B[v2])은 디스크에 기록됨**.
  이 경우에 파일 시스템 메타데이터의 일관성은 보장된다. 아이노드의 포인터는 블록 5번을 가리키고 비트맵은 5번이 사용 중을 나타내므로 파일 시스템의 메타데이터의 관점에서 보면 모든 것이 괜찮아 보인다. 한 가지 문제는 5번 블록에는 의미없는 값이 들어 있다는 것이다[2].
- **비트맵(B[v2])을 제외한 아이노드(I[v2])와 데이터(Db)는 디스크에 기록됨**.
  이 경우에는 디스크의 데이터를 아이노드가 제대로 가리키고 있지만 이전 버전의 비트맵 (B1)과 아이노드 간의 내용이 일관성이 없다. 그러므로 파일 시스템을 새로 시작하기 전에 이 문제를 해결해야 한다.
- **아이노드(I[v2])를 제외한 비트맵(B[v2])과 데이터(Db)는 디스크에 기록됨**.
  이 경우는 아이노드와 비트맵 간의 내용이 일치하지 않는다. 블록이 기록되고 비트맵은 사용 중이라고 되어 있지만 아이노드가 파일을 가리키고 있지 않기 때문에 해당 블록이 어느 파일에 속한 것인지 알 길이 없다.

크래시 일관성 문제

크래시 때문에 파일 시스템 디스크 상의 자료 구조에 많은 문제가 발생할 수 있다는 것을 보았다. 파일 시스템 자료 구조 간의 불일치가 있을 수 있으며, 공간 누수가 발생할 수 있고, 사용자에게 의미없는 데이터가 전달되는 등의 여러 문제가 있다. 우리는 파일 시스템의 일관성이 **항상** 유지되도록 만들고자 한다. 연산 이전의 상태에서 연산 이후의

2) 역자 주. 이 상황은 문제가 없는 것일까? 만약 은행 계좌 이체 정보가 기록되어야 한다면, 결과는 어떨까?

상태로 이동할 때, 중간의 임시 상태 (transient state)로 파일 시스템이 남아 있는 경우를 방지하는 것이 목적이다. 불행하게도 디스크는 한 번에 하나의 쓰기 작업을 처리할 수 있으며 이러한 작업들 중간에 크래시나 전력 손실이 발생할 수 있어서 목적을 쉽게 달성할 수가 없다. 이러한 일반적인 문제를 **크래시 일관성 문제**라고 부른다(**일관된 갱신 문제**라 부르기도 한다).

45.2 해법 1: 파일 시스템 검사기

초기의 파일 시스템은 크래시 일관성을 해결하기 위해 간단한 방법을 사용하였다. 기본적으로 파일 시스템이 일관성이 없더라도 그대로 두었다가 리부팅 시에 일관성 문제를 해결하는 방식을 선택하였다. **fsck**[3]라고 하는 도구가 이 접근 방식의 고전적인 예이다. `fsck`는 일관성 불일치를 발견하고 수정하는 UNIX 도구다 [MK96]. 다른 시스템에도 디스크 파티션을 검사하고 고치는 도구들이 있다. 다만 이 방식이 문제들을 전부 해결할 수 없다는 것에 유의해야 한다. 위에서 살펴보았던 경우들 중에 아이노드가 의미없는 블럭을 가리키고 있지만, 파일 시스템은 일관성이 있는 것처럼 보였던 경우가 그런 예이다. 이 도구들의 목적은 파일 시스템 메타데이터들 간의 일관성을 유지하는 것이다.

McKusick과 Kowalski의 논문 [MK96]에서 말하고 있듯이 `fsck`의 동작은 여러 단계로 구성된다. 이것은 파일 시스템이 마운트 *직전*에 실행된다(`fcsk` 실행 시에는 파일 시스템이 어떤 동작도 실행하지 않다는 것을 가정한다). 종료가 되면 디스크 상의 파일 시스템이 일관성을 갖게 되며 사용자가 사용할 수 있게 된다.

`fsck`가 하는 기본적인 일을 아래와 같이 정리할 수 있다.

- **슈퍼블럭:** `fsck`는 먼저 슈퍼블럭 내용에 오류가 없는지를 검사한다. 대부분의 검사는 파일 시스템에 블럭 개수가 파일 시스템의 크기보다 더 큰지와 같은 기초 검사들로 이루어져 있다. 이러한 검사를 통해 (손상이) 의심되는 슈퍼블럭이 있는지 찾아내는 것이다. 발견하면 시스템은(또는 관리자는) 슈퍼블럭을 사본으로 대체할지를 결정한다.
- **프리 블럭:** 그 다음으로 `fsck`는 아이노드와 간접 블럭, 이중 간접 블럭 등을 살펴보고 파일 시스템에 현재 어떤 블럭들이 할당되었는지에 대한 정보를 생성한다. 얻은 정보를 토대로 정확한 할당 비트맵을 재구성한다. 기존의 비트맵과의 일치하지 않으면 아이노드 정보를 기반으로 불일치를 해결한다. 모든 아이노드에 대해서 같은 검사를 수행하여 아이노드 비트맵의 유효성을 검사하고, 필요 시 아이노드 비트맵을 재구성한다.
- **아이노드 상태:** 각 아이노드가 손상되었는지 다른 문제는 없는지 검사한다. 예를 들어 `fsck`는 각 할당된 아이노드가 유효한 속성(예, 일반 파일, 디렉터리, 심볼릭 링크 등)을 갖고 있는지 확인한다. 만약 아이노드의 항목 중에 쉽게 해결이 불가능한 문제가

3) ‘‘에프-에스-씨-케이” 또는 “에프-에스-체크”로 발음되며 이 도구가 마음에 들지 않으면 “에프-썩(엉망)”이라고 부를 수 있다. 진지한 전문가들이 이런 용어를 쓴다.

존재하면, `fsck`는 해당 아이노드를 의심 대상으로 간주하고 초기화한다. 아이노드 비트맵도 그에 따라 갱신된다.

- **아이노드 링크:** `fsck`는 각 할당된 아이노드의 링크 개수를 확인한다. 기억하겠지만, 링크의 개수는 특정 파일에 대한 참조를(즉, 링크) 포함하고 있는 디렉터리들의 수를 나타낸다. 링크 개수를 확인하기 위해서 `fsck`는 루트 디렉터리를 시작으로 모든 디렉터리 트리를 탐색하여 파일 시스템의 모든 파일과 디렉터리에 대한 링크 개수를 직접 수집한다. 새롭게 계산된 개수와 아이노드에서 확인한 수와 다른 경우가 있다면, 일반적으로 아이노드의 개수 필드를 고치는 방식으로 수정한다. 만약 할당된 아이노드는 있지만 어떤 디렉터리도 이를 참조하지 않는다면, 그 파일은 `lost+found` 디렉터리로 이동된다.
- **중복:** `fsck`는 중복된 포인터가 있는지도 검사한다. 즉, 같은 블럭을 가리키는 서로 다른 아이노드가 있는지를 검사한다. 만약 한 아이노드가 누가봐도 불량이면 초기화한다. 대안으로는 참조되고 있는 블럭의 사본을 만들어서 원하는 대로 각 아이노드가 하나씩 자신의 블럭을 가리키도록 하는 방법도 있다.
- **배드 블럭:** 모든 포인터 목록을 검사하면서 배드 블럭 포인터들도 함께 검사한다. 어떤 포인터가 "배드"라고 판단되면 당연히 그 포인터가 유효하지 않는 공간을 참조하고 있다는 것을 말한다. 예를 들면, 파티션 영역을 넘어서는 곳의 주소를 갖고 있는 경우이다. 이와 같은 경우에 `fsck`는 다른 해결 방법이 없기 때문에 아이노드나 간접 블럭에서 해당 포인터를 단순히 삭제(초기화)한다.
- **디렉터리 검사:** `fsck`는 파일의 내용을 파악하는 것은 불가능하다. 하지만 디렉터리 내용에 대해서는 파일 시스템이 생성한 구체적이고 서식화된 정보가 있으므로 `fsck`는 각 디렉터리의 내용에 대해서는 추가적으로 모든 내용이 제대로 저장되어 있는지의 검사를 수행한다. 디렉터리의 첫 항목이 "."과 ".."인지 디렉터리의 각 아이노드가 실제 할당되어 있는지 그리고 전체 계층에서 디렉터리가 두 번 이상 연결된 경우는 없는지를 검사한다.

보는 바와 같이 제대로 작동하는 `fsck`를 만드는 것은 보통일이 아니다. 파일 시스템 전반에 대한 이해가 필요하다 [SGu+08]. `fsck`(그리고 유사한 접근법들)는 보다 근본적인 문제점이 존재한다. `fsck`는 너무 느리다는 것이다. 아주 큰 디스크 볼륨의 경우 디스크 전체에서 할당된 모든 블럭들을 찾고 디렉터리 트리 전부를 읽어내는 것은 몇 분에서 수 시간이 걸릴 수도 있다. 디스크의 용량이 커지고 RAID가 대중화되면서 `fsck`는 실질적으로 사용이 불가능할 정도로 느리다(최근의 발전에도 불구하고 [Ma+13])[4].

상위 계층에서 보면 `fsck`의 메커니즘은 사실 매우 이상하다. 위에서 예로 들었던 세 개의 블럭만 디스크에 기록하는 경우를 생각해 보자. 세 개의 블럭을 갱신하다 생긴 문제를 해결하기 위해서 디스크 전체를 다 읽어본다는 것은 엄청난 비용이다. 이 상황은 마치 침대방 바닥에 열쇠를 흘리고 *지하실로부터 시작해서 아래서 위로 모든 방을 다*

4) 역자 주: 금요일 오후 다섯시, 회사 영업이 끝나고 서버의 fsck를 시작하였다. 월요일 아침 09:00 영업시작 시간까지 fsck가 종료되지 않을 수 있다.

돌아다니며 집 전체를 뒤져서 열쇠를 찾는 알고리즘과 유사하다. 가능하긴 하지만 낭비이다. 디스크(그리고 RAID)의 용량이 커지면서 연구자들과 실무자들은 다른 해법을 찾기 시작했다.

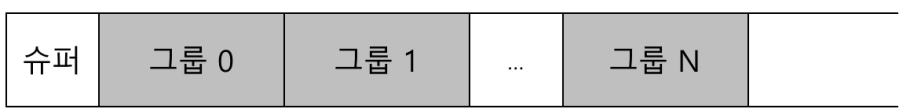
45.3 해법 2: 저널링(또는 Write-Ahead Logging)

일관성을 담보하는 가장 대중적인 해법은 데이터베이스 관리 시스템에서 차용한 개념 중 하나다. **WAL**이라고 알려져 있으며 정확히 이러한 문제를 해결하기 위해서 만들어졌다. 파일 시스템에선 write-ahead logging을 역사적인 이유로 **저널링(journaling)**이라고 부른다. 이를 처음 적용한 파일 시스템은 Cedar [Hag87]이다. 현대에 와서는 많은 시스템들이 이 개념을 사용하고 있다. 그 중에는 Linux의 ext3와 ext4, reiserfs, IBM의 JFS, SGI의 XFS, 그리고 Windows의 NTFS가 있다.

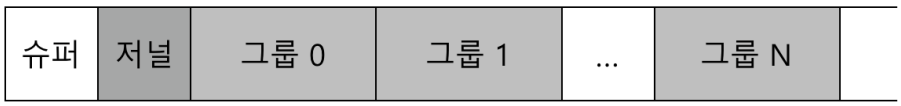
기본 개념은 다음과 같다. 디스크 내용을 갱신할 때, 해당 자료구조를 갱신하기 전에, 먼저 수행하고자 하는 작업을 요약해서 기록해둔다(디스크의 다른 잘 알려진 위치에). 다양한 기록방법이 존재한다. 새로운 페이지 이미지 전체를 저장하기도 하고, 변경될 부분만 저장하기도 한다. 이렇게 앞으로 할일을 미리 저장해놓는 것을 "write-ahead(미리 쓰기)"라고 하고 "log(로그)"라는 자료 구조에 기록하기 때문에 WAL이라고 부른다.

디스크에 갱신할 값, 즉 갱신 후에 저장될 값과 관련된 내용을 로그에 기록해 놓았기 때문에, 안심이다. 해당 디스크 페이지들을 새값으로 (덮어 쓰기) 갱신하는 과정에서 크래시가 발생하면, 로그를 확인해서 다시 갱신하면 된다. 이를 redo라고 부르기도 한다. 자명한 이름이다. 이렇게 하면 크래시가 났을 때 디스크 전체를 다 스캔하지 않아도 어느 부분을 고쳐야 하는지 (그리고 어떻게 고쳐야 할지) 정확히 알 수 있다. 저널링 파일 시스템에서는 쓰기가 좀 더 복잡해진다. 쓸 내용들에 대한 정보를 미리 로깅을 한후에 실제 쓰기를 진행하기 때문이다. 쓰기가 약간 느려진다. 하지만, 그 오버헤드가 의외로 작다. 시스템의 복구 성능을 대폭 개선 하기 때문에 대부분의 파일 시스템이 채용하고 있다.

이제 실전이다. 많이 사용되는 저널링 파일 시스템인 **Linux ext3** 파일 시스템의 저널링 기법을 알아보자. ext3가 사용하는 디스크 자료 구조는 대부분 **Linux ext2** [Twe98]의 그것과 동일하다. 디스크는 블럭 그룹으로 나뉘어 있고 각 블럭 그룹은 아이노드 비트맵과 데이터 비트맵 그리고 아이노드와 데이터 블럭들로 구성이 되어 있다. 저널 자료 구조가 새로운 핵심 자료 구조다. 이것은 저장장치의 일부 영역 (예: 128 MByte) 을 차지한다. 파일 시스템 포맷 시에 설정한다. 저널이 없는 ext2 파일 시스템은 다음과 같이 구성되어 있다.



저널 영역이 동일한 파일 시스템 파티션에 있다면(저널 영역을 다른 장치나 혹은 다른 파일 시스템 파티션에 위치시키는 것도 가능하다.) 저널 영역을 포함하는 ext3 파일 시스템은 다음과 같이 구성된다.

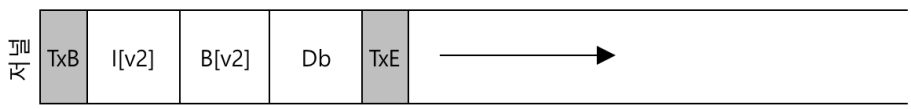


ext2와 ext3 간의 차이는 저널의 존재 여부와 그리고 당연히 저널의 활용 방법이다.

데이터 저널링

데이터 저널링의 동작 방법을 이해하기 위해 간단한 예제를 살펴보자. Linux ext3 파일 시스템에서 제공하는 저널 모드 중에 하나인 데이터 저널링을 중심으로 설명하겠다.

아이노드(I[v2])와 비트맵(B[v2]) 그리고 데이터 블럭(Db)을 갱신하는 평범한 파일 시스템 연산을 생각해 보자. 디스크의 최종 위치에 각각을 기록하기 전에 로그(또는 저널)에 이들을 먼저 쓴다. 로그는 다음과 같이 보일 것이다.



그림에서 보듯이 다섯 개의 블럭을 기록하였다. 트랜젝션의 시작 블럭(TxB)은 연산에 대한 정보들을 기록한다. 시작 블럭에 기록되는 정보로 갱신될 블럭들에 대한 정보 (예, I[v2]와 B[v2] 그리고 Db의 최종 주소)와 **트랜젝션 식별자**(**TID**)와 같은 것이 있다. 디스크 상의 최종 위치에 기록될 내용들이 가운데 세 개의 블럭에 그대로 기록되어 있다. 갱신해야 할 물리적 내용을 저널에 기록하기 때문에 **물리 로깅**으로도 불린다(대안으로는 갱신에 명령어 자체를 저장하는 **논리 로깅**도 있다. "이 갱신은 파일 X의 데이터 블럭 Db에 데이터를 덧붙이려고 한다"와 같은 내용이 로그에 기록된다. 좀 복잡하다. 그러나 로그 공간을 줄일 수 있으며 성능을 개선할 수 있는 여지도 있다). 마지막으로 트랜젝션 종료 블럭(TxE)은 이 트랜젝션의 종료를 알리며 마찬가지로 TID를 포함하고 있다. 트랜젝션 종료 블럭이 로그에 기록되면 트랜젝션은 "커밋(commit)"되었다고 말한다.

트랜젝션이 디스크에 안전히 기록된 후, 파일 시스템 상의 자료 구조들은 이제 갱신될 수 있다. 저널에 기록된 내용을 실제 위치에 반영하는 과정을 **체크포인팅**이라 부른다. 파일 시스템 **체크포인트** 시에(즉, 파일 시스템의 상태를 저널에 기록되어 있는 갱신 정보에 따라 최신으로 만드는 것) 앞에서 본 것과 같이 디스크의 원래의 위치에 I[v2]와 B[v2] 그리고 Db를 쓰도록 요청한다. 이 쓰기 요청이 성공적으로 완료되면 파일 시스템을 성공적으로 체크포인팅한 것이고 모든 동작이 끝난 것과 마찬가지다. 작업의 순서는 다음과 같다. 저널에 기록한 직후에 바로 체크포인트를 진행하지는 않는다.

1. 저널 기록: 트랜젝션을 저널에 기록한다. 구체적으로 트랜젝션 시작 블럭, 갱신될 데이터 블럭와 메타데이터 블럭, 그리고 트랜젝션 종료 블럭을 로그에 기록한다. 이 블럭들이 디스크에 안전하게 기록될 때까지 대기한다.
2. 체크포인트: 갱신된 메타데이터와 데이터 블럭들을 해당 위치에 반영한다.

예제에서는 TxB와 I[v2]와 B[v2]와 Db 그리고 TxE를 저널에 기록하였다. 각 쓰기가 완료되면 I[v2]와 B[v2] 그리고 Db를 각각의 디스크의 최종 위치에 체크포인팅하여 갱신 과정은 종료한다.

저널에 기록하는 도중에 크래시가 발생하게 되면 일이 복잡해진다. 예제에서는 트랜젝션(예, TxB, I[v2], B[v2], Db, TxE)에 속한 블럭 집합을 디스크에 쓰려고 한다. 이를 달성하는 간단한 방법은 하나씩 요청을 디스크에 전달하고 각각이 완료되기를 기다렸다가 다음을 요청하는 것이다. 하지만 이 방법은 너무 느리다. 이상적인 상황은 다섯 개의 요청을 한꺼번에 전송해서 디스크가 이들을 차례로 순차 쓰기로 하는 것이다. 매우 효율적일 것 같지만, 문제가 존재한다. 한번에 많은 양을 쓰고자 할 경우, 디스크가 스케줄링을 통해 임의로 이들을 작은 단위로 나누어 기록할 가능성이 존재한다. 그리고 이들의 완료순서는 디스크에 의해 결정되며, 파일 시스템이 의도했던 순서와 완전히 다를 수 있다. 예를 들어, 디스크 내부적으로는 (1) TxB, I[v2], B[v2] 그리고 TxE를 쓰고 (2) 나중에 Db를 쓸 수도 있다. 불행하게도 디스크의 전원이 (1)과 (2) 중간에서 차단된다면 디스크의 상태는 다음과 같이 된다.

뭐가 문제일까? 저널링에서 트랜젝션 종료 블럭은 매우 중요한 역할을 한다. 트랜젝션 종료 블럭이 기록되면, 트랜젝션의 성공적 종료를 의미한다. 우리의 경우, 데이터 블럭이 저널에 기록이 되지 않았음에도 불구하고 트랜젝션이 성공적으로 정상 종료된 것처럼 보인다는 것이다(시작과 끝에 동일한 식별자 순서 번호를 갖고 있다). 저널의 네 번째 블럭 내용을 파일 시스템이 정상적인 내용인지 아닌지를 알 수가 없다. 비트맵이나 아이노드 블럭은 그 내용을 살펴보면 값들이 의미가 있는 값인지 아닌지를 대략이라도 예상할 수 있다. 저널 트랜젝션의 네 번째 블럭 위치에는 파일의 데이터가 기록되어야 한다. 블럭의 내용을 읽어도 잘못된 블럭인지 아닌지를 알 수가 없다. 시스템이 재부팅을 하면 복구를 수행한다. 로그 영역을 검사하여, 저널에 기록된 내용들을 제 위치에 체크포인트 할 것이다. 위의 경우에는 "??"라고 쓰여진 쓰레기 Db 블럭을 실제 위치에 그대로 복사하게 된다. 사용자의 데이터일 때도 문제지만, 슈퍼블럭과 같이 파일 시스템의 핵심 자료구조가 이런 식으로 잘못 복구되면 마운트조차도 할 수 없게 된다. 최악의 시나리오다.

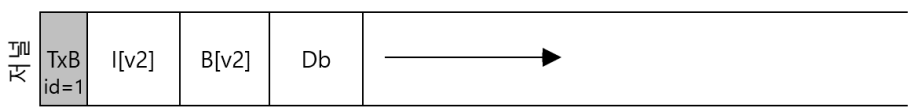
여담: 디스크에 강제로 쓰기

두 번의 디스크 쓰기 간의 순서를 보장하기 위해서 현대의 파일 시스템은 각별히 주의를 기울이고 있다. 과거에는 `A`와 `B`로 이루어진 두 번의 쓰기의 순서를 보장하는 일은 어렵지 않았다. 디스크에 `A`를 쓰기 요청하고 쓰기가 완료되면 운영체제에 인터럽트를 건다. 그 후에 `B`에 대한 쓰기를 요청했다.

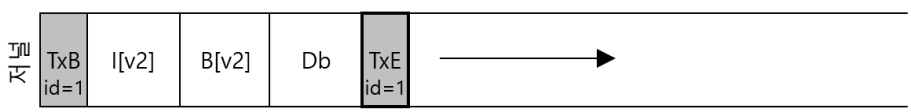
디스크 내에 있는 쓰기 캐시의 사용이 늘어나면서 상황이 좀 더 복잡해졌다. 쓰기 버퍼링이 동작 중이면(때로는 **즉시 보고**(**immediate reporting**)라고도 불림) 데이터는 디스크의 메모리 캐시에만 있고 디스크에는 다 쓰여지지 않았는데도 운영체제에게 쓰기가 완료했다고 알린다. 이때 두 번째 쓰기를 운영체제가 요청하면 첫 번째 쓰기 다음에 두 번째 쓰기가 디스크에 도달한다고 보장할 수 없기 때문에 쓰기 간의 순서는 보존이 안 된다. 한 가지 방법은 쓰기 버퍼링을 끄는 것이다. 하지만, 좀 더 최신의 시스템은 좀 더 주의를 기울여서 명시적으로 **쓰기 배리어**(**write barrier**)를 요청한다. 배리어 요청이 완료되었을 때는 어떤 쓰기 요청이든 배리어 이전에 요청된 것들은 배리어 요청 이후에 쓰기 요청을 받은 것들보다 먼저 디스크에 도달하도록 한다.

이 모든 기술들은 디스크가 제대로 동작한다는 강한 믿음을 바탕으로 한다. 하지만, 최근의 연구에 따르면 몇몇 디스크 제조사들은 명시적으로 **쓰기 배리어** 요청을 하더라도 "더 빠른 성능"을 위해서 그 명령을 무시하여 마치 디스크가 빠르게 동작하는 것처럼 보이게 만든다. 하지만, 그렇게 되면 결과적으로는 부정확하게 동작할 수 있다는 위험을 갖고 있다고 한다 [Chi+13; Raj+11]. Kahan이 말했듯이 빨라서 오류가 생긴다 하더라도 거의 언제나 느린 것보다 빠른 것이 좋다.

이러한 문제의 발생을 방지하기 위해 파일 시스템은 트랜젝션을 두 단계로 나누어 기록한다. 먼저 TxE를 제외한 모든 블럭을 한 번의 쓰기 요청으로 저널에 쓴다. 완료되면 저널은 다음과 같이 보인다(덧붙이는 워크로드였다고 가정하자).



이 쓰기가 완료되면 파일 시스템은 TxE 블럭에 대한 쓰기를 요청하여 저널을 최종적이고 안전한 상태로 만든다.



매우 중요한 조건이 충족되어야 한다. 디스크 쓰기 연산의 원자성이다. 트랜젝션 종료 블럭(TxE)는 무조건 원자적으로 기록되어야 한다. 디스크는 섹터단위(512바이트) 크기의 쓰기에 대한 원자성은 보장한다. 하드디스크에 한 섹터, 즉 512 바이트를 기록할

여담: 로그 쓰기의 최적화

로그 쓰기의 비효율성을 깨달았을 것이다. 먼저 파일 시스템은 명시적으로 트랜젝션 시작 블럭을 쓰고 트랜젝션의 내용을 쓴 후에야 트랜젝션 끝 블럭을 디스크에 쓴다. 디스크가 어떻게 동작하는지 생각해 보면 성능에 미치는 영향은 분명해진다. 대부분 한 번의 추가 회전이 발생한다(왜 그런지 생각해 보라).

졸업생 중 하나인 Vijayan Prabhakaran은 간단한 개념으로 이 문제를 해결하였다 [Pra+05]. 트랜젝션의 시작과 끝 블럭에 저널 내용에 대한 체크섬을 포함하도록 하였다. 그러면 파일 시스템이 전체 트랜젝션을 기다지 않고 한 번에 쓸 수 있도록 해 준다. 복구 시에 파일 시스템이 계산한 트랜젝션의 체크섬과 기록되어 있는 체크섬이 불일치한다면 트랜젝션 기록 중에 크래시가 있었다고 결론을 내릴 수 있으며 파일 시스템을 갱신하지 않도록 할 수 있다. 그러므로 쓰기 규약과 복구 시스템을 약간 변경하면 파일 시스템은 일반적인 경우들에서 빠른 성능을 얻을 수 있다. 더 나아가 저널의 내용을 읽을 때도 체크섬으로 보호를 받기 때문에 파일 시스템은 신뢰성을 갖게 된다.

이 간단한 수정은 Linux 파일 시스템 개발자들 사이에서 충분한 관심을 끌었고, 그 내용을 차세대 Linux 파일 시스템인 (예상했듯이) **Linux ext4**에 적용하였다. 안드로이드 휴대 기기 플랫폼을 포함하여 세계의 수백만 대의 기기에서 사용 중이다. Linux 기반의 시스템에서 디스크에 무언가 쓸 때마다 Wisconsin에서 개발한 작은 코드가 당신의 시스템을 조금 더 빠르고 신뢰성 있게 해 주고 있다.

경우, 512 바이트중 첫 256바이트는 기록되고 뒤 256바이트는 누락되는 그런 경우는 발생하지 않는다. TxE 쓰기의 원자성을 보장하기 위해서는 하나의 512 바이트 블럭에 들어가도록 만들어야 한다. 지금까지 다룬 과정을 정리해 보자.

1. **저널 쓰기:** 트랜젝션의 내용을 로그에 쓴다(TxB와 메타데이터 그리고 데이터를 포함함). 그리고 이 쓰기가 완료되길 기다린다.
2. **저널 커밋:** 트랜젝션 커밋 블럭을 로그에 쓴다(TxE를 포함). 그리고 트랜젝션은 커밋됨이라고 한다.
3. **체크포인트:** 갱신(메타데이터와 데이터)한 내용을 디스크 상의 최종 위치에 쓴다.

복구

크래시 상황에서 저널을 사용하여 파일 시스템을 **복구**하는 방법을 살펴보자. 크래시는 파일 시스템을 갱신하는 과정에서 어느 시점에서든 발생할 수 있다. 만약 트랜젝션이 로그에 안전하게 기록되기 전에 크래시가 발생한다면(즉, 위에서 말한 2단계가 끝나기 전에), 할 일은 간단하다. 복구시에 아무것도 안하면 된다. 트랙잭션이 로그에 기록되었지만, 체크포인트가 완료되기 전에 크래시가 발생한다면, 다음과 같은 방식으로 **복구**할 수 있다. 파일 시스템의 복구 프로세스는 시스템이 부팅할 때 로그를 탐색해서 디스크에 커밋된 트랜젝션이 있는지 파악한다.

커밋된 트랜젝션의 블럭들을 디스크 상의 원래의 위치에 쓴다. 이 과정을 **재생**(**replayed**)한다. 가장 간단한 방식의 로깅이다. **redo logging**이라고 한다. 저널에 커밋된 트랜젝션을 replay하여 디스크 자료 구조 간에 일관성을 보장한다. 복구 후 파일 시스템을 마운트하여 새로운 요청을 받을 수 있도록 준비한다.

체크포인트 중에는 어느 시점에서든 크래시가 발생해도 문제가 없다는 것을 기억하자. 블럭들이 일부만 최종 위치에 반영되어도 상관없다. 최악의 경우라고 해봐야 복구 시에 해당 갱신 작업을 다시 수행하는 것이다. 복구는 자주 발생하는 작업이 아니기 때문에 (갑작스러운 시스템 크래시가 있은 후에 수행되기 때문에), 몇 번의 쓰기를 더 한다고 문제 될 것은 없다[5].

로그 기록을 일괄 처리 방식으로

이제까지 언급한 데이터 저널링 방식은 디스크에 엄청나게 많은 트래픽을 유발한다. 같은 디렉터리에서 두 개의 파일 `file1`과 `file2`를 연속으로 생성했다고 해 보자. 파일을 하나 생성하려면 디스크 상의 여러 자료 구조를 갱신해야 한다. 최소한 아이노드 비트맵 (새로운 아이노드를 할당), 새롭게 생성된 파일의 아이노드, 새로운 디렉터리 항목을 포함하는 부모 디렉터리의 데이터 블럭, 그리고 부모 디렉터리의 아이노드(새롭게 변경된 시간을 갖고 있음)가 갱신된다. 저널링을 사용하면 각 파일들에 대한 정보들이 모두 저널에 커밋한다. 두 파일이 같은 디렉터리 내에 존재한다면, 파일 아이노드들도 같은 아이노드 블럭에 존재할 가능성이 크다. 같은 블럭을 계속 반복해서 기록할 수 있다.

이런 상황을 개선하기 위해서 어떤 파일 시스템은 각각 여러 개의 저널로그를 모아서 한 번에 디스크에 커밋하는 방법을 사용한다(예, Linux ext3). 로깅 해야 할 모든 파일 시스템 갱신 내용을 트랜젝션 버퍼라는 자료 구조에 보관한다. 파일 시스템 파티션 마다 하나만 존재하는 전역 자료구조이다. 저널 버퍼라고 불리기도 한다. 위의 예제와 같이 두 개의 파일이 생성되는 경우 메모리에 존재하는 아이노드 비트맵, 파일들의 아이노드들, 디렉터리 데이터, 그리고 디렉터리 아이노드를 "갱신됨"(dirty)으로 설정하고 해당 블럭들을 트랜젝션 버퍼 리스트에 추가한다. 트랜젝션 버퍼의 내용들은 파일 시스템이 저널을 커밋할 때 디스크에 기록된다. 파일 시스템은 메모리에 있는 갱신내용들을 정기적으로 (예: 5초, 이 간격을 저널 타임아웃이라 부르기도 한다.) 저널에 기록한다. 이외에 `fsync()`나 `sync()`함수가 호출되면, 해당 시점에 트랜젝션 버퍼의 내용들이 저널에 커밋된다. 저널에 기록되어야 할 블럭들을 버퍼링함으로써 많은 양의 쓰기 트래픽을 줄일 수 있다.

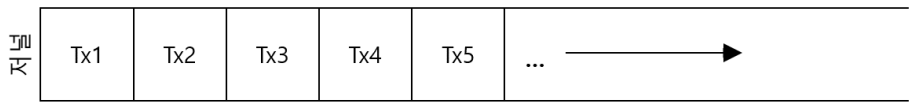
로그 공간의 관리

파일 시스템의 디스크 상의 자료 구조를 갱신하는 기본적인 방법을 정리하였다. 파일 시스템은 갱신의 내용을 메모리에 일정 시간 동안 버퍼에 유지한다. 갱신 내용을 디스크에

5) 모든게 다 걱정스럽다면, 우리도 어떻게 도울 방법이 없다. 너무 걱정하지 말자, 건강에 해롭다! 근데 이제는 너무 걱정하는 것에 대한 걱정을 하고 있는지도 모르겠다.

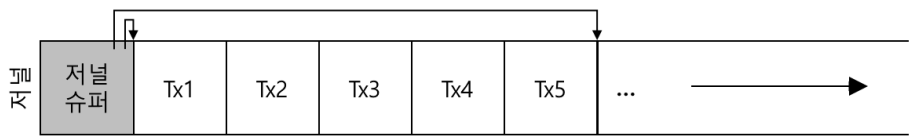
기록할 시점이 되면, 먼저 파일 시스템은 저널에 저널 트랜젝션의 상세 내용을 쓴다 (write-ahead logging). 트랜젝션이 완료되면 파일 시스템은 각 블럭들을 디스크의 최종 위치에 체크포인트한다.

로그의 크기는 정해져있다. 트랜젝션이 계속 추가되면 (그림에서와 같이) 얼마지나지 않나 공간이 소진될 것이다. 그러면 어떻게 될까?



로그공간이 가득 차면 두 가지 문제가 발생한다. 첫 번째 문제는 간단하고 그렇게 치명적이지는 않다. 로그에 있는 모든 트랜젝션을 (순서대로) 재실행해야 하기 때문에 로그가 커질수록 복구 소요 시간은 길어진다. 두 번째 문제는 심각하다. 만약 로그가 가득차면(또는 거의 찼다면) 디스크에 더 이상의 트랜젝션을 커밋할 수가 없게 된다. 파일 시스템을 갱신하는 모든 작업들이 실패한다. 파일 시스템이 쓸모없게 된다.

이 문제를 해결하기 위해서 저널링 파일 시스템은 로그를 환형 자료 구조로 형식으로 사용한다. 로그영역을 끝까지 다 쓰면, 앞에서부터 다시 쓰기 시작한다. 이와 같은 이유로 저널을 때로는 **환형 로그**(**circular log**)라고도 부른다. 파일 시스템은 트랜젝션이 체크포인트되면 해당 로그 공간이 재사용될 수 있도록 트랙젝션이 차지하고 있던 공간을 비운다. 이 기능은 다양한 방법으로 구현할 수 있다. 예를 들어 최신 트랜젝션과 가장 오래된 트랜젝션의 위치를 **저널 슈퍼블럭**에 기록해 놓는다. 두 영역 사이의 블럭들은 의미있는 로그 정보를 가지고 있으며, 그 외의 공간은 재사용 가능한 빈 공간이 된다. 이 방식을 그림으로 표현하면 아래와 같다.



저널 슈퍼블럭은(파일 시스템의 슈퍼블럭과 혼돈하면 안 된다) 체크포인트가 안 된 트랜젝션들을 구분할 수 있을 만큼의 충분한 정보를 갖고 있다. 이 정보를 통해 복구 시간을 단축시키고, 로그를 재사용한다. 기본 저널 알고리즘에 다음의 과정을 추가한다.

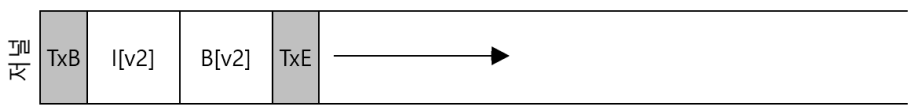
1. **저널 쓰기:** 트랜젝션의 내용을 로그에 쓴다(TxB와 갱신에 대한 내용을 포함). 이 쓰기가 완료되기를 기다린다.
2. **저널 커밋:** 트랜젝션 커밋 블럭을 로그에 쓴다(TxE를 포함). 쓰기가 완료되기를 기다린다. 이제 트랜젝션은 **커밋**된 상태이다.
3. **체크포인트:** 갱신에 대한 내용을 파일 시스템 내의 원래의 위치에 쓴다.
4. **프리:** 일정 시간 이후에 저널 슈퍼블럭을 갱신하여 저널의 트랜젝션을 해제한다.

앞의 절차가 최종 데이터 저널링 프로토콜이다. 하지만 아직 문제가 있다. 크래시는 사실 별로 자주 일어나지 않는다. 복구를 위해 모든 데이터 블럭을 디스크에 *두 번씩* 기록하는 것은 여러모로 너무 부담스럽다. 데이터를 두 번씩 쓰지 않으면서 일관성을 유지할 수 있는 방법을 생각해 낼 수 있겠는가?

메타데이터 저널링

복구 시간 단축은 어느 정도 성공했지만(디스크 전체를 스캔하는 것에 비해 저널을 읽고 몇 개의 트랜젝션을 재실행함), 파일 시스템이 느려졌다. 디스크로 내려가는 모든 쓰기가 저널에 먼저 기록되어야 하기 때문에 쓰기 양이 두 배가 된다. 순차쓰기 워크로드는 특히 치명적이 될 수 있다. 디스크 대역폭의 반만 쓸 수 있기 때문이다. 더 나아가 저널에 쓰는 것과 주 파일 시스템에 쓰는 것 사이에는 탐색 비용도 존재한다. 특정 워크로드의 경우에 심각한 양의 쓰기 오버헤드가 추가될 수 있다.

모든 데이터 블럭을 두 번씩 쓰는 방법을 배제하기 위한 기법들이 존재한다. 이제까지 다룬 저널링 모드는 모든 데이터를(파일 시스템의 메타데이터와 함께) 저널링하기 때문에 **데이터 저널링**이라고 부른다(Linux ext3에서처럼). 간단한(그리고 좀 더 대중적인) 저널링의 형태는 **Ordered journaling**이다(또는 **메타데이터 저널링**이라고 함). 저널에 데이터 블럭을 기록하지 *않는다*는 것을 제외하면 거의 대부분이 동일하다. 앞에서 수행했던 것과 동일한 갱신작업을 수행하면 다음과 같은 정보가 저널에 기록된다.



이전에는 데이터 블럭 Db가 로그에 기록었지만 Ordered 모드 저널링에서는 추가적인 쓰기를 피하기 위해 파일 시스템의 원래 위치에 Db를 기록한다. 디스크로 내려가는 I/O 트래픽의 대부분이 데이터인 것을 감안하면 두 번씩 쓰지 않는 것만으로도 저널링으로 내려가는 I/O의 오베헤드의 정도를 상당히 감소시킬 수 있다. 새로운 Ordered 모드 저널링에서는 데이터 블럭을 언제 디스크로 내려보내야 할까?

이 문제에 대한 이해를 돕기 위해서 파일에 블럭을 추가하는(allocating write) 예제를 다시 살펴보자. 갱신 대상은 I[v2]와 B[v2] 그리고 Db로 세 개의 블럭이다. 첫 두 개의 블럭은 메타데이터이다. 로그에 기록된 후, 나중에 체크포인트된다. 세 번째 블럭은 파일 시스템에 한 번만 기록이 된다. Db를 언제 디스크에 내려 보내야 할까? 그리고 내려 보내는 시점은 중요할까?

메타데이터만 저널링하는 경우 데이터 블럭을 디스크에 내려 보내는 시점이 매우 중요하다. 예를 들어보자. I[v2]와 B[v2]가 저널에 커밋된 후에, 디스크에 Db를 기록해도 될까? 문제가 있다. 파일 시스템은 일관성 있는 것처럼 보이지만, I[v2]가 쓰레기 데이터를 가리킬 수 있다. I[v2]와 B[v2]는 저널에 기록되었지만, Db가 디스크에 아직 기록되지 않은 상황을 생각해 보자. 크래시가 발생했다. 파일 시스템이 복구를 시도할

것이다. Db가 로그에 기록되지 *않았기* 때문에 파일 시스템은 I[v2]와 B[v2]를 재실행하여 체크포인트하고 파일 시스템이 일관성을 유지하도록 할 것이다(파일 시스템의 메타데이터 입장에서). 하지만 I[v2]는 쓰레기 데이터를 가리키고 있다. 즉, Db가 저장되었어야 할 위치에 있는 예전 값을 가리키고 있다.

어떻게 이 상황을 해결할 수 있을까? 답은 의외로 간단하다. 일반적인 파일 시스템(예, Linux ext3)에서는 메타데이터를 저널에 기록하기 전에 반드시 관련 데이터 블럭들을 디스크에 *먼저* 쓴다. 구체적인 절차는 다음과 같다.

1. **데이터 쓰기:** 데이터를 최종 위치에 쓴다. 완료될 때까지 대기한다(사실 꼭 대기할 필요는 없다. 상세 설명은 아래를 참고하기 바란다).
2. **저널 메타데이터 쓰기:** 시작 블럭과 메타데이터를 로그에 쓴다. 완료될 때까지 기다린다.
3. **저널 커밋:** 트랜젝션 커밋 블럭을 로그에 쓴다(TxE를 포함). 쓰기가 완료될 때까지 기다린다. 트랜젝션은 이제 (데이터를 포함하여) **커밋**된 상태이다.
4. **체크포인트 메타데이터:** 갱신된 메타데이터의 내용을 파일 시스템 상에 있어야 할 최종 위치에 갱신한다.
5. **해제:** 저널 슈퍼블럭에 해당 트랜젝션이 해제되었다고 표기한다.

데이터가 먼저 기록되는 것을 강제하여 아이노드 포인터가 쓰레기 데이터를 가리키지 않는 것을 보장한다. 크래시 일관성에 있어서 핵심 법칙은 "포인터의 대상이 되는 객체를 그것을 가리키는 객체보다 먼저 써라"는 것이며 일부 크래시 일관성 기법들에서는 더 넓은 의미로 사용되고 있다 [GP94](아래의 상세 설명을 참고).

대부분의 시스템에서 메타데이터 저널링(ext3의 정렬된 저널링(ordered journaling)류)이 데이터 저널링 보다 훨씬 대중적으로 사용된다. 예를 들어, Windows의 NTFS와 SGI의 XFS는 둘 다 메타데이터 저널링의 일종을 사용한다. Linux ext3에서는 데이터, Ordered 또는 Unordered 모드를 선택할 수 있다(Unordered 모드에서는 데이터가 임의의 시점에 기록될 수 있다). Ordered 모드와 Unordered 모드는 둘 다 메타데이터 일관성은 보장하지만, 데이터의 일관성에 대한 보장은 다르다. Unordered 모드의 경우는 아이노드가 쓰레기 데이터 블럭을 가리킬 수 있기 때문에 데이터의 일관성은 보장되지 않는다.

저널링의 올바른 동작에 있어서, 저널 쓰기(2 단계) 이전에 데이터 쓰기(1 단계)가 반드시 완료될 필요는 없다. 데이터 쓰기 요청, 저널 트랜젝션 시작 블럭, 그리고 메타데이터를 함께 요청해도 문제 없다. 하지만, 저널 커밋 블럭을 요청(3 단계)하기 전에 1 단계와 2 단계는 반드시 완료되어야 한다.

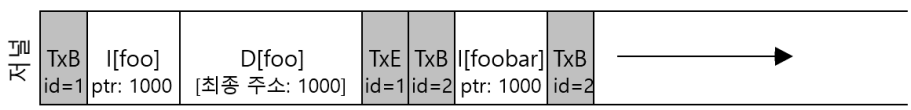
까다로운 사례: 블럭 재활용

저널링에 있어서, 몇 가지 특수한 경우들을 다루도록 하겠다. 블럭 재활용과 관계 있다. Stephen Tweedie는 이렇게 말했다(ext3 배후의 주요 권력자 중 한 명).

> "전체 시스템에서 가장 신경 쓰이는 동작은 무엇일까? ... 파일 삭제이다. 삭제 과정에서 발생하는 작업들은 주의를 요한다. 파일 삭제를 위한 모든 구체적인 작업들... 블럭 삭제와 재할당은 신경을 상당히 많이 써야 한다." [Twe00]

Tweedie가 제시한 예제는 다음과 같다. 만약 메타데이터 저널링을 사용한다고 가정한다. 데이터 블럭들은 저널링을 안 한다. `foo`라는 디렉터리가 있다. 사용자는 `foo` 디렉터리에 파일을 생성하였다. 디렉터리에 항목이 추가된다. `foo`의 데이터 블럭이 로그에 기록된다. 디렉터리는 메타데이터로 간주되기 때문이다. `foo` 디렉터리의 데이터 블럭은 1000번에 위치한다고 해 보자. 저널링이 완료되면 로그는 다음과 같은 정보를 갖고 있게 된다.

이 시점에서 사용자가 디렉터리의 모든 파일을 디렉터리 자체와 함께 삭제하였다. 블럭 1000번은 free 블럭이 된다. 사용자가 새로운 파일(`foobar`라고 하자)을 생성하였다. `foo`가 사용하던 삭제된 블럭 1000번을 할당받게 되었다. 데이터와 함께 `foobar`의 아이노드는 디스크에 저장된다. 단, 메타데이터 저널링이 사용되고 있기 때문에 `foobar`의 아이노드만 저널에 커밋된다는 것을 유의해야 한다. 블럭 1000번에 새롭게 써진 `foobar`의 데이터 블럭은 저널에 기록되지 않았다.



이제 크래시가 발생했다고 가정하자. 복구 시 로그에 있는 모든 정보를 순서에 따라 다시 실행한다. 이때 블럭 1000번에 있었던 디렉터리 데이터 정보를 체크포인트한다. 복구 과정에서 사용자 데이터인 현재의 `foobar` 파일은 이전의 디렉터리 정보로 덮어써지게 된다. 이전 값으로 복구가 되었다. 사용자가 `foobar` 파일을 읽으면 엉뚱한 값이 읽혀진다.

이 문제는 다양한 해결법이 있다. 그 중의 하나는 지워진 블럭이 체크포인트될 때까지 절대로 재사용하지 않는 것이다. Linux ext3은 저널에 **철회** 레코드(**revoke** record)라는 새로운 항목을 추가하였다. 위의 경우처럼 디렉터리를 삭제하면 저널에 철회 레코드를 기록하도록 만들었다. 저널을 재실행 시 시스템은 먼저 철회 레코드의 존재 여부를 먼저 탐색하고, 철회된 내용은 재실행을 하지 않는 식으로 위의 문제를 회피하였다.

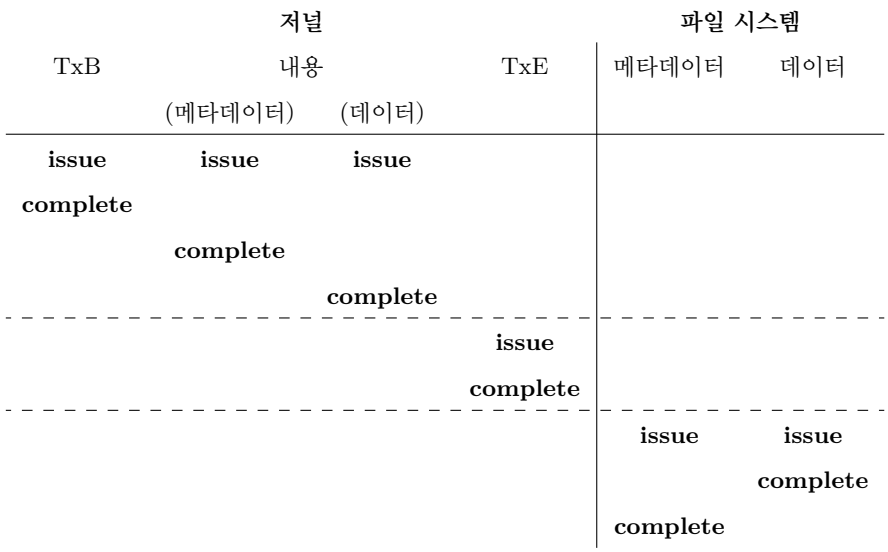
마치며: 흐름도

저널링에 대한 논의를 끝내기 전에 논의된 쓰기 작업들을 시간순으로 정리해 보자. 그림 45.1은 데이터와 메타데이터를 같이 저널링하는 프로토콜을 나타내고 그림 45.2는 메타데이터만 저널링하는 프로토콜을 나타낸다.

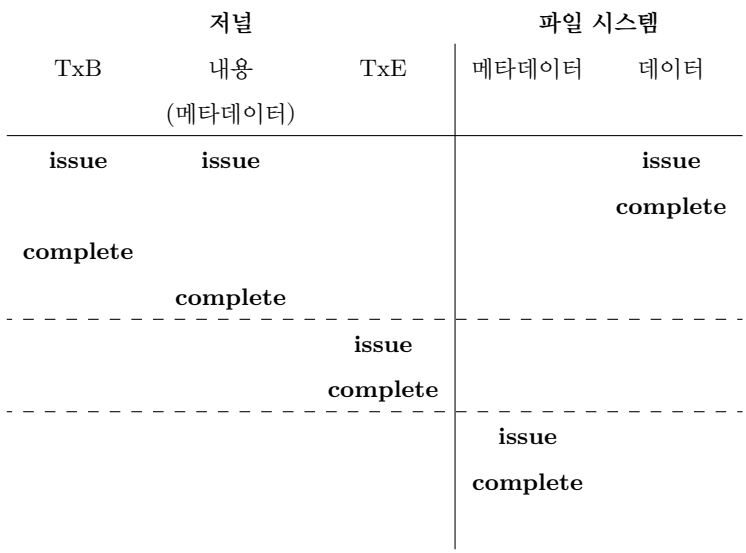
그림에서 시간의 흐름은 아래 방향이고 그림의 각 줄은 쓰기가 요청되거나 완료가 될 수 있는 논리적 시간을 나타낸다. 예를 들어, 데이터 저널링 프로토콜에서(그림 45.1) 트랜잭션의 시작 블럭(TxB)과 트랜잭션의 내용 블럭들은 논리적으로 동시에 요청될 수 있고, 이들 간의 완료 순서에 특별한 제약이 없다. 하지만 트랜잭션 끝 블럭(TxE)은 앞선 쓰기들의 완료보다 절대로 먼저 완료되면 안 된다. 이 순서를 보장하기 위해 TxE의 쓰기 요청은 앞선 쓰기 요청들이 모두 완료된 후에 시작해야 한다. 마찬가지로, 데이터와 메타데이터 블럭을 체크포인트하는 쓰기는 트랜잭션 끝 블럭이 커밋되기 전까지는 시작될 수 없다. 수평 점선은 쓰기 순서가 반드시 유지되어야 하는 시점을 나타낸다.

메타데이터 저널링 프로토콜에서도 흐름도는 비슷하다. 데이터 블럭과 저널의 트랜잭션 시작 블럭은(TxB) 동시에 쓰기 요청될 수 있음을 유의해야 한다. 데이터 쓰기는 트랜잭션 끝(TxE)의 쓰기가 요청되기 이전에 완료되어야 한다.

마지막으로 흐름도에 나타낸 각 쓰기의 완료 시간은 이론적으로 예측이 불가능하다는 것을 유의해야 한다. 실제 시스템에서 완료 시간은 하부 I/O 시스템(저장 단위)에 의해 결정된다. 저장 장치는 성능을 위해서 쓰기 순서를 재정렬할 수 있다. 유일하게 보장되는 것은 쓰기 연산들 간의 기본 순서 뿐이다(그림에서 수평 점선으로 나타나 있다).



〈그림 45.1〉 **데이터 저널링 시간 흐름표**



〈그림 45.2〉 **메타데이터 저널링 시간 흐름표**

45.4 해법 3: 그 외 방법

지금까지 파일 시스템의 메타데이터의 일관성을 유지하는 두 가지 방법을 설명하였다. 하나는 `fsck`를 사용하는 느린 방법이고, 다른 하나는 좀 더 효율적인 저널링이라는 기법이다. 하지만 이외에도 다른 기법들이 있다. 그 중에 하나는 Soft Update [GP94]로 알려진 것으로 Ganger와 Patt에 의해 소개되었다. 이 방법은 파일 시스템의 모든 쓰기들의 순서를 잘 정해서 디스크 상의 자료 구조가 절대로 불일치 상태가 되지 않도록 한다. 예를 들면, 데이터 블럭을 가리키는 아이노드가 기록되기 *전에* 그 아이노드가 가리키는 대상 데이터 블럭을 디스크에 먼저 기록하여 아이노드가 쓰레기 값을 가진 블럭을 가리키지 않도록 만들 수 있다. 유사한 종류의 법칙을 파일 시스템의 모든 자료 구조에 적용할 수 있다. 하지만, Soft Update는 구현이 쉽지 않다. 저널링 기법은 파일 시스템의 자료 구조에 관한 상대적으로 적은 지식만으로도 구현이 가능하지만, Soft Update는 각 파일 시스템 자료 구조에 대한 심도 있는 지식이 필요로 하다. 시스템이 훨씬 복잡해진다.

다른 접근 방법은 **Copy-On-Write**(**COW**)라고 불리는 것으로 Sun의 ZFS를 포함하여 여러 대중적인 파일 시스템에서 사용되는 방법이다 [BM07]. 이 기술은 파일이나 디렉터리들을 절대로 원래 위치에 덮어쓰지 않는다. 대신에 이제까지 디스크에서 사용 안 된 위치에 갱신 내용을 저장한다. 여러 개의 갱신 작업이 완료되면 COW 파일 시스템은 루트 자료 구조에 새롭게 갱신된 자료 구조를 가리키는 포인터를 포함하도록 변경한다. 이러한 방식으로 파일 시스템의 일관성을 유지하도록 한다. 앞으로 다루게 될 로그 기반 파일 시스템(LFS)의 장에서 이 기술에 대해서 논의하게 될 것이다. LFS는 COW의 초기 예 중 하나이다.

또 다른 방법은 최근에 Wisconsin에서 개발한 것이다. **백포인터 기반 일관성**(**backpointer-based consistency**, **BBC**)이라고 하는 기법으로 쓰기 사이에 어떤 순서도 강요하지 않는다. 대신 시스템의 모든 블럭에 **백포인터**를 추가하였다. 예를 들면 각 데이터 블럭은 자신이 속해 있는 아이노드에 대한 참조를 갖고 있는 식이다. 파일을 접근할 때 이 파일이 일관성이 유지되고 있는지를 판단하기 위해서는 앞 방향 포인터(예, 아이노드 내의 주소 또는 직접 블럭)가 가리키는 블럭이 다시 이 블럭을 가리키고 있는지를 검사하면 된다. 만약 참이면 모든 것이 디스크로 안전하게 도달하였다는 것이고 파일이 일관성을 갖고 있음을 나타낸다. 만약 아니라면 파일은 불일치 상태이며 에러를 반환한다. 파일 시스템에 백포인터를 추가하여 크래시 일관성을 보장한다 [Chi+12].

마지막으로 우리는(Univ. of Wisconsin Remzi 교수 연구팀) 저널링 시 디스크 쓰기를 대기하는 횟수를 줄이는 기법을 연구하였다. **Optimistics Crash Consistency** [Chi+13]라는 이름의 새로운 기법은 요청할 수 있는 가장 큰 크기의 쓰기를 디스크로 내려 보낸다. 그리고 **트랜젝션 체크섬**(**transaction checksum**) [Pra+05]과 다른 몇 가지 기술들을 적용하여 일관성을 보장한다. 어떤 워크로드에서는 이 기술로 성능을 수십 배 이상 개선할 수가 있었다. 하지만 제대로 동작하려면 새로운 디스크 인터페이스가 필요하다 [Chi+13].

45.5 요약

크래시 일관성의 문제를 소개하고, 이 문제를 해결하는 다양한 접근법들을 설명하였다. 오래된 방법으로 파일 시스템 검사기를 만드는 것이 있기는 하지만 현대의 시스템에서는 복구 시간이 너무 느리다는 문제가 있다. 많은 파일 시스템들이 저널링을 사용한다. 저널링은 복구 시간을 디스크 볼륨 크기에서 로그의 크기로 줄이기 때문에 복구에 드는 소요 시간을 상당히 줄일 수가 있다. 현대의 많은 파일 시스템은 저널링을 사용한다. 저널링이 여러 다른 형태로 존재하는 것도 살펴보았다. 가장 보편적으로 사용되는 것은 Ordered 모드 저널링이다. 이 기법은 파일 시스템의 메타데이터뿐 아니라 사용자 데이터까지 적절한 수준의 일관성을 유지하도록 보장하면서 저널로 내려가는 트래픽의 양을 줄이는 기법이다.

참고 문헌

[BM07] **"ZFS: The LastWord in File Systems"**
Jeff Bonwick and Bill Moore
URL: `http://opensolaris.org/os/community/zfs/docs/zfs%20last.pdf`
ZFS는 기록 중 복사(COW)와 저널링을 사용한다. 사실 어떤 경우에는 디스크에 쓰기를 로깅하는 것의 성능이 더 좋을 때가 있다.

[Chi+13] **"Optimistic Crash Consistency"**
Vijay Chidambaram, Thanu S. Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau
SOSP '13, Nemacolin Woodlands Resort, PA, November 2013
좀 더 낙관적이고 성능이 좋은 저널링 프로토콜을 다룬 우리의 연구이다. `fsync()`*를 많이 호출하는 워크로드에서 성능이 크게 개선될 수 있다.*

[Chi+12] **"Consistency Without Ordering"**
Vijay Chidambaram, Tushar Sharma, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau
FAST '12, San Jose, California
백포인터를 기반으로 하는 새로운 형태의 크래시 일관성을 다룬 우리의 최근 연구이다. 흥미로운 내용을 담고 있으니 읽어보라!

[GP94] **"Metadata Update Performance in File Systems"**
Gregory R. Ganger and Yale N. Patt
OSDI '94
일관성을 달성할 수 있는 여러 방법 중의 하나로 쓰기의 순서를 잘 정하도록 한 현명한 논문이다. 나중에 BSD 기반 시스템에 구현되었다.

[Hag87] **"Reimplementing the Cedar File System Using Logging and Group Commit"**
Robert Hagmann
SOSP '87, Austin, Texas, November 1987
Write-ahead logging(또는 저널링으로 알려짐)을 파일 시스템에 적용한 첫 연구이다(우리가 알고 있는).

[Ma+13] **"ffsck: The Fast File System Checker"**
Ao Ma, Chris Dragga, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau
FAST '13, San Jose, California, February 2013
최근의 우리 연구로 fsck의 속도를 몇 배 이상 개선하기 위한 내용을 담고 있다. 이 중의 몇 가지 개념은 BSD 파일 시스템 검사기[MK96]에 이미 적용이 되었으며 현재 배포되었다.

[MK96] **"FSCK - The Unix File System Check Program"**
Marshall Kirk McKusick and T. J. Kowalski
*종합적인 파일 시스템 도구를 다룬 첫 연구이다. 논문과 도구의 이름이 동일하다. FFS [***MJLF84b***]를 만든 사람들 중 일부가 이 도구를 개발하였다.*

[PAA05] **"Analysis and Evolution of Journaling File Systems"**
Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau
USENIX '05, Anaheim, California, April 2005
저널링 파일 시스템의 동작을 분석한 우리의 초기 논문이다.

[Pra+05] **"IRON File Systems"**
Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau
SOSP '05, Brighton, England, October 2005
디스크 오류에 파일 시스템이 어떻게 동작하는지를 주로 연구한 논문이다. 끝에 가서는 트랜젝션 체크섬을 소개하면서 로깅 속도를 개선하였다. 그리고 이후에 Linux ext4에 적용되었다.

[Raj+11] **"Coerced Cache Eviction and Discreet-Mode Journaling"**
Abhishek Rajimwale, Vijay Chidambaram, Deepak Ramamurthi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau
DSN '11, Hong Kong, China, June 2011
디스크의 문제를 다룬 우리의 논문이다. 명시적으로 메모리가 아닌 디스크에 쓰라고 했는데도 불구하고 디스크에 쓰지 않고 메모리 캐시에 쓰기를 버퍼로 갖고 있는 문제이다. 우리가 제안한 이 문제의 해결책은 이렇다. 디스크에 B 이전에 A를 먼저 쓰기 원한다면, A를 먼저 쓰고 디스크에 "가짜" 쓰기를 대량으로 보내는 것이다. 바라건대 메모리에 공간을 만들기 위해서 A가 디스크에 강제로 써지게 될 것이다. 비실용적이라는 것을 빼면 괜찮은 해법이다.

[SGu+08] **"SQCK: A Declarative File System Checker"**
Haryadi S. Gunawi, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, and RemziH. Arpaci-Dusseau
OSDI '08, San Diego, California
SQL 질의문을 사용하여 새롭고 좀 더 나은 파일 시스템 검사기를 만든 우리의 논문이다. 기존의 검사기가 갖고 있는 문제점들을 나타내었으며 fsck의 복잡도의 직접적인 결과인 여러 버그들과 이상 행동을 찾아내었다.

[Twe00] **"EXT3, Journaling Filesystem"**
Stephen Tweedie
URL: `olstrans.sourceforge.net/release/OLS2000-ext3/OLS2000-ext3.html`
Tweedie의 ext3에 대한 연설 기록문이다.

[Twe98] **"Journaling the Linux ext2fs File System"**
Stephen C. Tweedie
The Fourth Annual Linux Expo, May 1998
Tweedie는 Linux ext2 파일 시스템에 저널링을 추가하는 어려운 일을 대부분 했다. 그 결과로 놀랍지 않겠지만, ext3이 탄생하였다. 괜찮은 설계 결정 중에는 하위 호환성에 강하게 집중한 것도 포함된다. ext3로 사용하기 위해서 기존의 ext2 파일 시스템에 저널링 파일을 단순하게 추가하고, ext3 파일 시스템으로 마운트하면 된다.