46

로그 기반 파일 시스템

1990년대 초반에 Berkeley 대학의 John Ousterhout 교수와 그의 대학원생 Mendel Ronsenblum이 로그 기반 파일 시스템(Log-structured File System)이라는 새로운 파일 시스템을 개발하였다 [RO91]. 다음과 같은 관찰결과가 연구의 동기가 되었다.

- **메모리 크기가 증가 추세였다:** 메모리 용량이 커지고 더 많은 데이터를 메모리에 캐시할 수 있게 되었다. 더 많은 데이터가 캐시되면서 디스크의 사용은 대체적으로 쓰기 위주가 되었고 읽기는 캐시에서 처리되었다. 파일 시스템의 성능이 쓰기의 성능에 의해 결정되었다.
- **임의 I/O와 순차 I/O의 성능 간의 격차가 크게 벌어졌다:** 전송 대역폭은 매해 대략 50%-100%가량 개선되었지만 탐색과 회전 지연 비용은 매우 천천히 감소하여 연 5%-10% 가량씩 개선되었다 [Pat98]. 디스크를 순차적으로 사용할 수 있다면 상당한 성능 개선 효과를 얻게 된다.
- **많은 일반적인 워크로드에서 기존 파일 시스템들의 성능이 좋지 않았다:** 예를 들어, FFS [McK+84a]는 블럭 하나 크기의 새로운 파일을 생성하기 위해서 여러 번의 쓰기를 해야 한다. 새로운 아이노드를 위해서 한 번, 아이노드 비트맵을 위해서 한 번, 파일이 들어 있는 디렉터리의 데이터 블럭을 위해서 한 번, 디렉터리 아이노드를 갱신하기 위해서도 한 번, 새 파일의 부분이 될 데이터 블럭을 위해서 한 번, 그리고 할당된 데이터 블럭을 표시하기 위해 데이터 비트맵을 쓴다. FFS는 이 모든 블럭들을 같은 블럭 그룹 내에 배치시킬 수는 있다. 그렇지만 FFS는 여러 번의 짧은 탐색과 연이은 회전 지연을 발생시키기 때문에 최대 순차 대역폭이 낼 수 있는 성능에 한참 못 미치는 성능을 보인다.
- **파일 시스템이 RAID를 고려하지 않았다:** 예를 들어 RAID-4와 RAID-5는 **작은 크기의 쓰기(small write)** 문제를 갖고 있다. 하나의 블럭을 쓰더라도 실질적으로 4번의 물리적 I/O가 발생하는 문제가 있다. 기존의 파일 시스템들은 이러한 RAID에서의 최악의 현상을 회피하기 위한 노력을 하지 않았다.

이상적인 파일 시스템은 쓰기 성능에 초점을 둔다. 디스크의 순차 대역폭 성능을 최대한 활용하는 방향으로 설계된다. 데이터 기록 뿐만 아니라 디스크 상의 메타데이터

구조를 빈번하게 갱신하는 워크로드에서도 효율적으로 동작한다. 마지막으로 단일 디스크뿐만 아니라 RAID에서도 잘 동작한다.

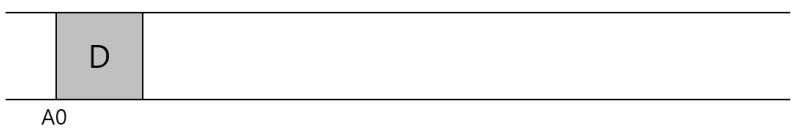
Rosenblum과 Ousterhout가 생각한 새로운 종류의 파일 시스템은 **로그 기반 파일 시스템**(**Log-structured file systems**)또는 짧게 **LFS**라고 불린다. 디스크 기록시, LFS는 모든 갱신 정보를 (메타데이터를 포함한) **세그멘트**라 불리는 메모리 자료구조에 보관한다. 세그멘트가 가득차면, 디스크에서 빈 공간을 찾아, 한번에 기록한다. LFS는 기존의 내용을 덮어 쓰지 않는다. 대신 *항상* 비어 있는 곳에 세그멘트들을 쓴다. 세그멘트가 크기 때문에 디스크 기록 작업이 효율적이된다. 디스크 성능을 최대한 활용할 수 있다.

핵심 질문: 모든 쓰기를 어떻게 순차 쓰기로 변형시킬까

어떻게 하면 모든 쓰기를 순차 쓰기로 변형시킬 수 있을까? 읽기 대상이 디스크 상에 흩어져 있을 가능성이 있기 때문에, 읽기동작을 순차적으로 변형하는 것은 원천적으로 불가능하다. 하지만, 쓰기의 경우 파일 시스템이 쓰기 위치를 선택할 수 있다. 선택의 여지를 최대한 활용해 보자.

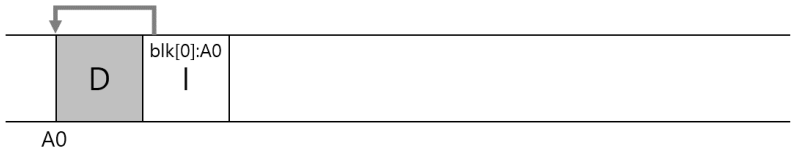
46.1 디스크에 순차적으로 쓰기

우리가 처음 직면한 문제는 다음과 같다. 파일 시스템을 변경하는 모든 쓰기 작업을 어떻게 순차 쓰기로 디스크에 보낼 수가 있을까? 이해를 돕기위해 간단한 예제를 살펴보겠다. 파일에 데이터 블럭 `D`를 쓴다고 생각해 보자. 데이터 블럭을 디스크에 쓰게 되면 디스크 상의 배치가 다음과 같이 될 것이다. 디스크의 주소 `A0`에 `D`가 쓰였다.



데이터 블럭을 기록하면 데이터만 갱신되는 것이 아니다. **메타데이터**도 같이 갱신되어야 한다. 이 경우 해당 파일의 **아이노드 (I)**도 디스크에 기록하고, 아이노드가 데이터 블럭 `D`를 가리키도록 한다. 데이터 블럭과 아이노드를 디스크에 함께 기록하면 다음과 같이 된다(실제로는 그렇지 않지만, 아이노드가 데이터 블럭만큼 크게 보이는 것에 유의하자. 대부분의 시스템에서는 데이터 블럭은 4 KB의 크기를 갖고 아이노드는 그보다 훨씬 작은 128바이트 정도의 크기를 갖고 있다).

모든 갱신(데이터 블럭과 아이노드 등)을 디스크에 순차적으로 기록한다는 것이 LFS의 핵심이다. 이것을 이해하였다면 기본 개념을 정복한 것이다. 하지만, 모든 복잡한 시스템이 그렇듯이 어려운 문제들은 각론에서 발생한다.



팁: 세부적인 것들이 의미가 있다

모든 흥미로운 시스템들은 몇 개의 일반적인 개념과 여러 개의 세부 정보들로 이루어져 있다. 때로는 이러한 시스템들을 배울 때면 "대충 감이 잡히는데, 그 이상은 세부 정보일 뿐이야"라고 생각할 수 있다. 그러면 당신은 실제 어떻게 동작하는지에 대해서 반만큼만 배운 것이다. 이렇게 하지 말자! 대부분 세부적인 정보가 핵심이다. LFS에서도 보게 되겠지만 기본 개념은 이해하기 너무 간단하다. 하지만, 실제 동작하는 시스템을 만들려면 모든 종류의 까다로운 경우들을 고려해야 한다.

46.2 순차적이면서 효율적으로 쓰기

디스크에 순차적으로 쓰는 것(만)으로는 효율적인 쓰기를 보장할 수가 없다. 예를 들어 시간 `T`에 주소 `A` 위치에 하나의 블럭을 썼다고 해 보자. 그리고 잠시 기다렸다가 $T + \delta$ 시간에 디스크의 주소 $A + 1$ 위치(순차적 순서에 따라 다음 블럭 주소)에 쓰기를 하였다. 불행하게도 첫 쓰기와 두 번째 쓰기 사이에 디스크는 회전을 하였다. 두 번째 쓰기의 경우, 디스크에 기록이 행해지기 전에 플래터를 회전시켜야 한다(구체적으로, 만약 회전이 $T_{rotation}$ 만큼의 시간이 걸린다면, 디스크 표면에 두 번째 쓰기를 실행하기 전까지 $T_{rotation} - \delta$ 시간을 기다려야 한다). *순차적인* 순서로 쓰는 것 만으로는 디스크의 최대성능을 달성하기 어렵다. 다수의 순차쓰기를 (또는 하나의 큰 크기의 쓰기) 한 번에 디스크에 내려 보내야 빠른 쓰기 성능을 얻을 수 있다.

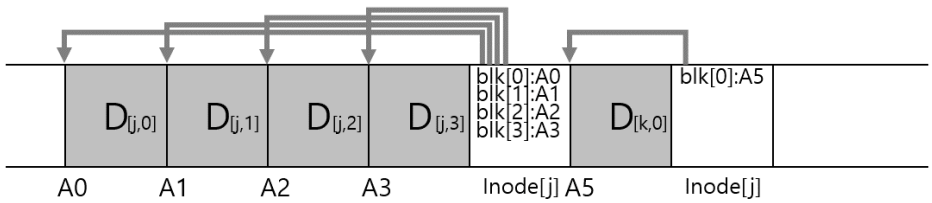
이를 위해 LFS는 **쓰기 버퍼링**[1]이라는 고전적인 기법을 사용한다. 디스크에 쓰기 전에 LFS는 갱신 내용을 메모리에 보관한다. 갱신된 내용이 충분히 쌓였다면 디스크로 한 번에 모두 내려 보내서 효율적인 디스크 작동을 도모한다.

LFS에서 한 번에 디스크에 기록하는 단위를 **세그멘트**라고 부른다. 세그멘트가 컴퓨터 시스템에서는 너무 자주 사용되는 용어이기는 하지만, 여기서는 LFS가 단일 집합으로 묶어서 쓰기 위해 사용하는 쓰기 단위를 말한다. 디스크에 데이터를 기록할 때 쓰기 내용들을 세그멘트 버퍼에 유지하고, 세그멘트가 차면 세그멘트 버퍼를 한 번의 쓰기 연산으로 디스크에 기록한다. 세그멘트가 충분히 크면, 쓰기연산은 효율적이 된다.

LFS가 두 개의 갱신 내용을 하나의 세그멘트 버퍼에 넣는 예제를 살펴보자. 실제 세그멘트는 이보다는 더 크다는 것을 참고하자(몇 MB단위). 첫 번째 추가된 갱신의 내용은 파일 j에 4개의 블럭을 쓰는 것이고, 두 번째로 갱신된 것은 파일 k에 한 개의

1) 사실 이 개념에 대해서 좋은 참고 문헌을 찾는 것이 상당히 어렵다. 컴퓨터는 역사 초기에 여러 사람들에 의해서 개발이 되었기 때문이다. 쓰기 버퍼링의 장점에 대한 연구로는 Solworth와 Orji의 연구 [SO90]를 참고하고 잠재적 손해에 대해서는 Mogul의 연구 [Mog94]를 참고하자.

블럭을 추가하는 것이다. LFS는 일곱 개의 블럭으로 이루어진 세그멘트 전체를 디스크로 한 번에 커밋한다. 최종적으로 디스크 상의 블럭들의 배치는 다음과 같다.



46.3 적절한 버퍼의 크기는?

그 다음 나오는 질문은 아래와 같다. 디스크에 실제 쓰기 전에 LFS는 몇 개의 쓰기 내용을 세그멘트 버퍼에 갖고 있어야 할까? 물론, 대답은 디스크의 물리적 특성에 의해 변한다. 전송 속도 대비 위치 잡기 오버헤드가 얼마나 큰지에 따라 달라진다. 이에 대한 분석은 FFS를 다뤘던 장을 참고하자.

예를 들어, 쓰기 시에 발생하는 위치 잡기(즉, 회전과 탐색 오버헤드)에 드는 시간이 $T_{position}$ 초가 걸린다고 하자. 디스크 전송 속도는 R_{peak} MB/s라고 하자. 적절한 세그멘트의 크기는 얼마일까?

매번 쓰기 시 디스크 헤드를 이동하는 데 일정 시간이 소요된다고 가정하자. 그러면, 위치 잡기 비용을 상쇄(**amortize**)하기 위해서는 얼마나 크게 써야 할까? 클수록 좋으며(당연하다), 최대 대역폭에 더 근접할 수 있다.

D MB 크기를 쓴다고 가정해 보자. 이 데이터 청크를 쓰는 데 소요되는 시간(T_{write})은 위치 잡기 시간 ($T_{position}$) 더하기 `D`를 전송하는 시간 ($\frac{D}{R_{peak}}$)이다. 또는,

$$T_{write} = T_{position} + \frac{D}{R_{peak}} \tag{46.1}$$

실제 쓰기 속도는 $R_{effective}$이며, 다음의 식과 같이 쓰인 데이터의 총량을 해당 데이터를 쓰는 데 소요된 총 시간으로 나눈 값을 나타낸다.

$$R_{effective} = \frac{D}{T_{write}} = \frac{D}{T_{position} + \frac{D}{R_{peak}}} \tag{46.2}$$

최대 속도에 근접하도록 유효 속도 ($R_{effective}$)를 구하고자 한다. 유효 속도가 최대 속도의 특정 비율(F)이 되도록 하는 것이다. F는 $0 < F < 1$의 값을 갖는다(대체로 F는 최대 속도의 0.9 또는 90%가 된다). 수학적으로는 $R_{effective} = F \times R_{peak}$ 식에 해당한다.

이제 D에 대해서 식을 정리해 보자.

$$R_{effective} = \frac{D}{T_{position} + \frac{D}{R_{peak}}} = F \times R_{peak} \tag{46.3}$$

$$D = F \times R_{peak} \times (T_{position} + \frac{D}{R_{peak}}) \tag{46.4}$$

$$D = (F \times R_{peak} \times T_{position}) + (F \times R_{peak} \times \frac{D}{R_{peak}}) \tag{46.5}$$

$$D = \frac{F}{1-F} \times R_{peak} \times T_{position} \tag{46.6}$$

예를 들어서 생각해 보자. 디스크 헤드 이동시간(위치잡기시간)이 10 ms이고 최대 전송 속도가 100 MB/s이다. 유효 대역폭이 최대 속도의 90%($F = 0.9$)가 되기를 원한다고 해 보자. 이 경우 $D = \frac{0.9}{0.1} \times 100\,\text{MB/s} \times 0.01\ \text{seconds} = 9\,\text{MB}$가 된다. 최대 대역폭에 도달하기 위해서는 얼만큼을 버퍼에 둬야 할지를 알아보기 위해 다른 수를 대입해 보자. 95%와 99%가 되려면 얼마나 되어야 할까?

46.4 문제: 아이노드 찾기

LFS에서 아이노드의 위치를 파악하는 방법을 알아보자. 먼저 일반적인 Unix 파일 시스템에서 아이노드를 찾는 방법을 살펴보자. FFS와 같은 파일 시스템이나 그보다 오래된 Unix 파일 시스템에서 아이노드를 찾는 것은 간단하다. 정해진 위치에 배열로 배치되어 있기 때문이다.

전통적인 파일 시스템에서는 각 아이노드의 위치가 정해져있다. 아이노드 번호와 시작 주소가 주어지면, 특정 아이노드의 위치는 아이노드 번호에 아이노드의 크기로 곱하고 배열의 시작 주소에 더하면 구할 수 있다. 배열을 사용하면 아이노드의 위치를 이와 같이 빠르고 간단하게 찾을 수 있다.

FFS에서 아이노드 번호를 사용하여 아이노드를 찾는 것은 좀 더 복잡하다. FFS는 아이노드 테이블을 분할하여 실린더 그룹마다 아이노드 그룹을 넣어 두기 때문이다. 분할된 테이블의 크기를 알아야 하고 각각의 시작 주소를 알아야 한다. 위치 계산은 배열 기반과 유사하며 마찬가지로 간단하다.

LFS의 경우는 좀 더 복잡하다. 아이노드가 디스크 전역에 흩어져 있기 때문이다. 뿐만 아니라, 원 위치에 덮어쓰기를 하지 않기 때문에, 최신 아이노드의 위치(즉, 우리가 원하는)가 계속 변하기 때문이다.

46.5 간접 계층을 이용한 해법: 아이노드 맵

계속적으로 이동하는 아이노드의 위치를 파악하기 위해 LFS 설계자들은 **아이노드 맵(inode map, imap)**이라는 자료 구조를 개발하였다. imap 자료 구조는 아이노드 번호를 입력으로 하여 가장 최신 아이노드의 디스크 위치를 구한다. 이 구조는 각 항목당 4 바이트(디스크 포인터)를 갖는 간단한 *배열*로 구현될 수 있다. 디스크에 아이노드가 기록될 때 imap은 새로운 위치를 가리키도록 갱신된다.

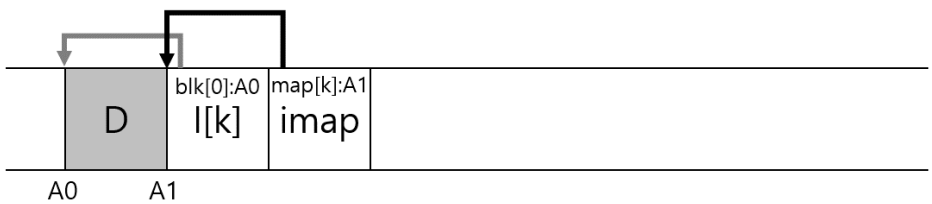
팁: 간접 계층을 이용하자

사람들은 검퓨터 과학에서 모든 문제의 해법은 **간접 계층**(**level of indirection**)의 도입이라고 한다. 분명한 것은 이것은 사실이 아니다. *대부분* 문제들의 해법일 뿐이다. 우리가 다뤘던 모든 종류의 가상화를 생각해 볼 수 있을 것이다. 예를 들어 가상 메모리가 단순히 간접 계층을 사용한다고 할 수 있을 것이다. LFS의 아이노드 맵도 분명히 아이노드 번호들의 가상화이다. 이와 같은 예제들에서 모든 참조들을 변경하지 않고도 자유롭게 자료 구조들(VM에서 페이지, 또는 LFS에서 아이노드)을 이동할 수 있게 하는 간접 참조의 능력을 볼 수 있기를 바란다. 물론, 간접 참조는 **오버헤드를 추가**한다는 단점도 갖고 있다. 다음에 문제를 만나면 간접 참조를 이용하여 해결해 보도록 하자. 하지만, 그 전에 사용할 때 발생하는 성능 오버헤드도 먼저 생각해 보자.

imap은 안전하게 보관되어야 한다(즉, 디스크에 써져야 한다). 그래야 LFS에 크래시가 발생하더라도 아이노드의 위치를 파악할 수가 있다. 다음 질문이 생긴다. imap은 디스크 어디에 저장해야 할까?

imap을 디스크의 고정된 위치에 배치하는 방법이 있다. imap은 자주 갱신된다. 파일 시스템의 내용이 변경될 경우, imap을 새로 기록해야 한다. 잦은 갱신으로 인해 성능이 저하될 수 있다(즉, 각 쓰기의 위치와 imap 간의 디스크 탐색이 더 많아지게 된다).

LFS에서는 아이노드 맵을 새로이 기록된 데이터와 아이노드들 옆에 함께 기록한다. 파일 k에 데이터 블럭을 추가할 때, LFS는 다음의 그림과 같이 새로운 데이터 블럭과 해당 아이노드 그리고 아이노드 맵의 일부분을 연속하여 디스크에 기록한다.



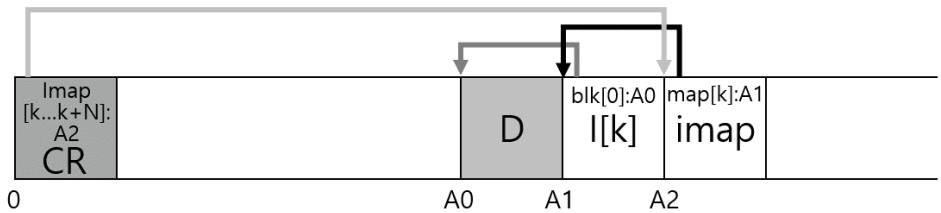
blk[0]:A0는 첫 번째 블럭의 주소가 A0라는 뜻이다. I[K]는 K 번째 아이노드를 나타낸다. map[K]:A1은 K 번째 아이노드가 A1에 위치해 있다는 뜻이다. 이 그림에서는 imap이라고 표기된 블럭에 저장되어 있는 아이맵의 일부가 아이노드 k는 디스크 주소 `A1`에 있다고 나타내고 있다. 이 아이노드는 데이터 블럭 `D`가 주소 `A0`에서 시작한다는 정보를 갖고 있다.

46.6 최종 완성: 체크포인트 영역

석연치 않은 점이 존재한다. 위의 방법에 의하면 아이노드 맵 역시 블럭으로 나누어져서 디스크 상에 흩어져 있게 된다. 아이노드 맵은 어떻게 찾을까? 특별한 비법은 없다. 정해진 위치에 검색을 시작하는 데 필요한 정보를 저장해야 한다.

LFS는 디스크 상에서 약속된 위치에 각 imap 블럭들의 위치를 기록한다. 이를 **체크포인트 영역**(**checkpoint region**, **CR**)이라고 부른다. 체크포인트 영역은 최신의 아이노드 맵을 이루는 블럭들을 가리키는 포인터(즉, 주소)를 갖고 있다. CR 영역을 읽어서 아이노드 맵의 조각들을 찾을 수가 있다. 체크포인트 영역은 주기적으로 갱신이 되기 때문에(약 30초 주기 등) 성능에 큰 악영향은 없다. 디스크 상의 파일 시스템 영역에 체크포인트 영역이 할당되어 있다(아이노드 맵의 최신 조각을 가리키고 있다). 일반적인 Unix 파일 시스템과 같이 아이노드 맵 조각들은 아이노드들의 주소들을 포함하며, 아이노드들은 파일들을(그리고 디렉터리들도) 가리킨다.

체크포인트 영역, 하나의 imap 블럭, 아이노드, 그리고 데이터 블럭으로 구성된 예제를 살펴보자(체크포인트 영역은 디스크의 가장 앞, 주소 0번에, 위치하고 있다). 실제 파일 시스템에서는 CR은 훨씬 더 크며(사실, 이제 곧 알게 되겠지만 두 벌을 갖고 있다), 더 많은 개수의 imap 청크, 아이노드, 데이터 블럭 등이 존재한다.



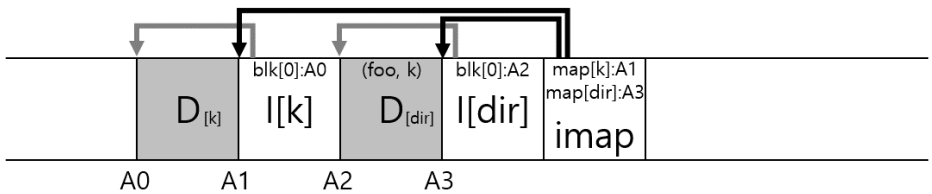
46.7 디스크에서 읽기: 요약

LFS가 동작하는 방식을 제대로 이해했는지 확인해 보자. 디스크에서 파일을 읽는 과정을 따라가 본다. 메모리에는 아무 것도 없다고 하자. 가장 처음 읽어야 하는 디스크 상의 자료 구조는 체크포인트 영역이다. 체크포인트 영역은 전체 아이노드 맵 블럭들을 가리키는 포인터들(즉, 디스크 주소들)을 갖고 있다. LFS는 아이노드 맵 전체를 읽어서 메모리에 캐시한다. 파일의 아이노드 번호를 구한다. LFS는 imap의 아이노드-디스크 주소 매핑에서 아이노드 번호를 찾아서 가장 최신의 아이노드를 읽어들인다. 이제 파일에서 블럭을 읽으려면 LFS는 일반적인 Unix 파일 시스템이 수행하는 동일한 절차를 밟는다. 직접 포인터를 따라가거나 간접 포인터 또는 이중 간접 포인터를 필요에 따라 따라간다. 보통의 경우 LFS는 일반 파일 시스템이 수행하는 개수 만큼의 I/O를 처리하여 파일을 읽는다. imap 전체가 캐시되어 있기 때문에 LFS가 추가로 하는 일은 imap에서 아이노드 주소를 찾아 읽는 것 뿐이다.

46.8 디렉터리 관리 방법은?

지금까지는 설명을 간단하게 하기 위해서 아이노드와 데이터 블럭들만 다루었다. 하지만 파일을 접근하려면(우리에게 친숙한 가짜 파일 이름인 `/home/remzi /foo`와 같은) 디렉터리들을 읽어야 한다. LFS는 어떻게 디렉터리 데이터를 저장할까?

디렉터리 자료 구조도 전통적인 UNIX 파일 시스템과 기본적으로 동일하다. 디렉터리는 매핑 정보(이름과 아이노드 번호)로 구성되어 있다. 파일을 생성할 때 LFS는 새로운 아이노드와 데이터 그리고 파일을 가리키는 디렉터리 데이터와 디렉터리 아이노드도 같이 써야 한다. LFS는 이 정보들을 디스크에 순차적으로 기록한다(일정 시간 동안 갱신 내용을 버퍼에 보관했다가). `foo`라는 파일을 디렉터리에 생성하기 위해 다음과 같은 자료 구조를 디스크에 추가 기록한다.



아이노드 맵 블럭은 디렉터리 파일 dir과 생성된 파일 `foo`의 아이노드 위치를 갖고 있다. `dir` 디렉터리에 존재하는 `foo`파일을 읽는 과정을 살펴보기로 하겠다. 먼저 아이노드 맵에서(대부분 메모리에 캐시됨) 디렉터리 *dir*의 아이노드 위치를 파악한다(A3). 그 후 디렉터리 데이터 (`A2`)의 위치를 얻기 위해 디렉터리 `dir`의 아이노드를 읽는다. 디렉터리의 데이터 블럭에서 (`foo`, k)의 파일명과 파일의 아이노드 번호 매핑 정보를 파악한다. 아이노드 번호 k (`A1`)의 아이노드 위치를 찾기 위해 아이노드 맵을 다시 한 번 조회한다. 최종적으로 주소 `A0`에 있는 데이터 블럭을 읽는다.

아이노드 맵은 **재귀 갱신 문제(recursive update problem)** [Zha+12]라는 LFS의 존재하는 심각한 문제를 해결한다. 이 문제는 원래의 위치에 덮어쓰기를 하지 않고 갱신 내용을 디스크의 새로운 위치에 갱신하는 파일 시스템(LFS와 같은)에는 모두 존재한다.

아이노드가 갱신되면 디스크 상의 아이노드 위치가 바뀐다. 만약 디렉터리의 항목들이 아이노드의 위치를 직접 가리키도록 설계되었다면, 아이노드의 위치가 변경되면 디렉터리의 데이터 블럭도 갱신되어야 한다. 이런 메커니즘 하에서는 디렉터리의 부모 디렉터리도 갱신해야 하고 또 그 위도 다 갱신하여 루트까지 올라가야 한다.

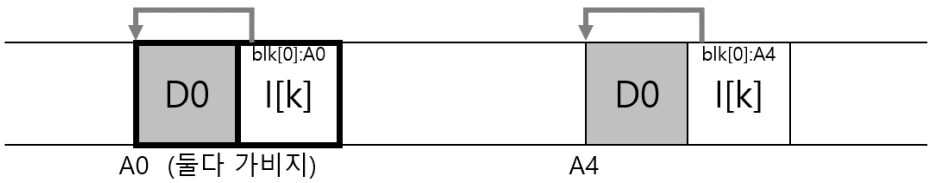
LFS는 이 문제를 imap을 사용하여 해결하였다. 아이노드 위치가 변경되더라도 변경 내용은 디렉터리 내에 직접 반영되지 않는다. 디렉터리에는 동일한 이름-아이노드 번호 매핑을 유지하면서 imap 자료 구조를 갱신한다. 이렇게 LFS는 간접 참조를 활용하여 재귀 갱신 문제를 해결하고 있다.

46.9 새로운 문제: 가비지 컬렉션

LFS에 다른 문제점이 존재한다는 것을 발견했을 것이다. LFS는 갱신된 파일(아이노드와 데이터)을 계속 디스크의 새로운 위치에 쓴다. 이 방법은 쓰기 동작을 효율적으로 수행하는 것이 목적이다. 하지만, 예전 값들이 디스크에 그대로 남아있게 된다. 사용되지는

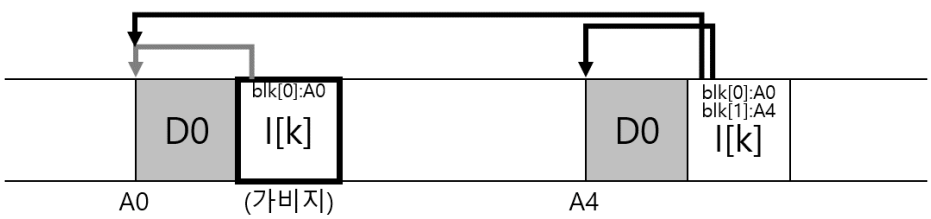
않지만 디스크의 공간을 차지하고 있다. 이 예전 내용들을 **가비지**(**garbage**, **쓰레기**)라고 부른다.

예로서 아이노드 번호 k가 기존 파일의 데이터 블럭 `D0`을 참조하는 경우를 생각해 보자. 우리가 데이터 블럭을 덮어쓰면 새로운 아이노드와 새로운 블럭이 생성이 된다. 그 결과로 디스크 상의 LFS 의 자료 구조는 다음과 비슷한 상태로 배치된다(간단히 표현하기 위해 imap과 다른 자료 구조들은 표현하지 않았다. 새 아이노드를 가리키는 새로운 imap 청크도 역시 디스크에 기록이 되어야 한다).



다음 그림을 보면 디스크에 두 버전의 아이노드와 데이터 블럭이 있는 것을 볼 수 있다. 하나는 이전의 것(좌측)이고 다른 하나는 현재 버전이다(우측). 데이터를 덮어쓰는 간단한 동작을 위해서 LFS는 여러 개의 새로운 자료 구조를 디스크에 기록한다. 이전 버전의 블럭들도 디스크에 여전히 존재한다.

또 다른 예제로 원래의 파일 k에 한 개의 블럭을 추가하는 경우를 생각해 보자. 이 경우 새로운 버전의 아이노드가 생성된다. 그렇지만, 예전의 데이터 블럭을 여전히 옛 버전의 아이노드가 가리키고 있다. 그러므로 내용은 여전히 남아있으며, 현재 파일 시스템의 일부로 남아있다.



구 버전의 아이노드와 데이터 블럭 등을 어떻게 해야 할까? 파일을 예전 버전으로 복원하는 데 사용할 수도 있다(실수로 파일을 덮어쓰거나 삭제한 경우에 유용해진다). 이와 같이 파일의 여러 버전을 관리하는 파일 시스템을 **버전 파일 시스템**(**versioning file system**)이라고 부른다.

하지만, LFS는 파일의 최신 버전만을 유지한다. LFS는 주기적으로(백그라운드로) 이전 버전의 데이터와 아이노드 그리고 다른 자료 구조들을 찾아 **제거**한다. 이 작업으로 디스크 블럭들을 해제하여 추후 쓰기 요청에 사용될 수 있도록 한다. 이 작업은 **가비지 컬렉션**의 일종이다. 가비지 컬렉션은 프로그램 언어에서 프로그램이 사용하지 않는 메모리를 자동적으로 해제하는 동작을 일컫는다.

앞에서 세그멘트를 설명할 때, 디스크에 큰 단위로 쓸 수 있기 때문에 LFS에서 세그멘트가 중요다고 하였다. 이제는 세그멘트가 효율적인 가비지 컬렉션 작업에도 필수적이라는 것을 알게 되었다. 만약에 LFS의 가비지 컬렉터(garbage collector)가 데이터 블럭과 아이노드 등을 하나씩 순회하며 해제한다면, 어떤 상황이 될지 생각해 보자. 결과는 이럴 것이다. 파일 시스템에는 할당 해제된 몇 개의 **구멍**(**hole**)과 할당된 디스크의 공간이 섞여 있는 상태가 될 것이다. 순차적으로 쓸 수 있는 연속적인 영역을 찾을 수가 없기 때문에 LFS의 쓰기 성능은 현저히 저하된다.

LFS의 가비지 컬렉터는 세그멘트 단위로 동작한다. 쓰기작업의 효율성을 위해 큰 공간 단위로 공간을 해제한다. 기본적인 작업 과정은 다음과 같다. LFS의 가비지 컬렉터는 주기적으로 오래된(일부분만 사용된) 세그멘트들을 읽은 후에 해당 세그멘트들에서 최신 블럭(유효한 내용을 갖는 블럭)들의 개수를 파악한다. 최신 버전 블럭들을 새로운 세그멘트로 이동한다. 유효 블럭들을 모두 이전한 후, 해당 세그멘트는 빈 공간으로 표시한다. 구체적으로 말하자면, 가비지 컬렉터는 M개의 기존의 세그멘트들을 읽은 후에, 해당 내용으로 N개의 새로운 세그멘트들에 채운다(이때 $N < M$ 이다, **compact**). 그리고 N개의 세그멘트를 디스크의 새로운 위치에 쓴다. 이전의 M개의 세그멘트들은 해제가 되며 파일 시스템이 추후 요청되는 쓰기들의 처리에 사용한다.

그래도 아직 두 개의 문제가 남았다. 첫 번째는 동작 메커니즘에 대한 것이다. LFS가 세그멘트 내에 어떤 블럭들이 살아 있고 죽은 것인지 어떻게 알 수 있을까? 두 번째는 정책에 대한 것이다. 가비지 컬렉터는 얼마나 자주 실행이 되어야 하고 몇 개의 세그멘트를 가비지 컬렉션해야 할까?

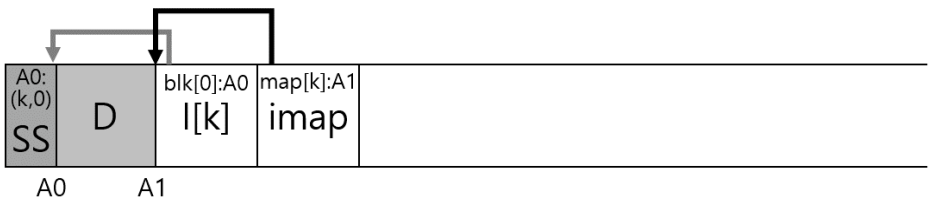
46.10 블럭의 최신 여부 판단

먼저 동작 메커니즘을 살펴보자. 세그멘트 `S`에 존재하는 데이터 블럭 `D`에 대해서, LFS는 `D`가 최신인지를 판단할 수 있어야 한다. 이를 위해 각 블럭에 대한 설명정보를 세그멘트에 추가한다. 구체적으로 살펴보자. LFS는 각 데이터 블럭 `D`에 대해 D가 속한 파일의 아이노드 번호(어떤 파일에 속하는지)와 파일 내에서 오프셋(해당 파일에 몇 번째 블럭인지)을 저장한다. 이 정보는 세그멘트의 첫 머리에 **세그멘트 요약 블럭**(**segment summary block**)이라 부르는 자료 구조에 기록된다.

이 정보를 활용하면 각 블럭의 유효성 여부를 파악할 수 있다. 디스크 주소 A에 위치한 블럭 `D`의 유효성 여부를 판단하는 과정을 알아보자. 세그멘트 요약 블럭에서 블럭 `D`의 아이노드 번호 N과 오프셋 T을 파악한다. 그 다음 imap에서 아이노드 N의 위치를 찾고, 디스크에서 그 N을 읽는다(어쩌면 이미 메모리에 올라와 있을 수도 있는데, 그러면 더 좋다). 마지막으로 아이노드(또는 어떤 간접 블럭)를 이용하여 오프셋 T에 해당하는 블럭의 디스크 위치를 알아낸다. 파악된 위치가 주소 A와 일치하면 블럭 `D`는 유효한 블럭이다. 만약 파악된 위치가 D의 위치와 다르다면, `D`는 유효하지 않다. 이 과정을 의사코드로 정리하면 다음과 같다.

```
(N, T) = SegmentSummary[A];
inode = Read(imap[N]);
if (inode[T] == A)
    // 블럭 D 는 유효함
else
    // 블럭 D 는 가비지
```

동작 원리를 그림으로 표현하면 다음과 같다. 주소 $A0$에 데이터 블럭이 있다. 세그멘트 요약정보에서 다음 사항을 얻을 수 있다. 데이터 블럭은 파일 k에 속한다. 해당 데이터 블럭은 파일 내에서 오프셋 0의 위치에 존재한다. 세그멘트에 존재하는 imap에서 아이노드 k의 위치를 파악할 수 있다.



복잡하다. 성능이 문제가 될 수 있다. 좀 더 빠른 유효성 판단 기법들이 있다. 예를 들면, 파일을 truncate하거나 삭제했을 때 LFS는 **버전 번호**(**version number**)를 증가시키고 그 새로운 버전 번호를 imap에 기록해 둔다. 버전 번호를 디스크 상의 세그멘트에 같이 기록해 두고 LFS는 디스크 상의 버전 번호와 imap의 버전 번호을 비교하여 위에서 설명한 긴 과정을 단축시킬 수가 있다. 그 결과 추가적인 읽기도 방지할 수 있다.

46.11 정책: 어떤 블럭을 언제 정리하는가

앞서 설명한 동작 메커니즘과 더불어 가비지 컬렉션 시기와 가비지 컬렉션 대상 블럭을 결정하는 정책이 필요하다. 가비지 컬렉션 시기를 정하는 것은 쉽다. 주기적으로 수행하는 방법이나 유휴 시간에 하는 방법 또는 디스크가 가득 차서 어쩔 수 없이 할 수밖에 없을 때 하면 된다.

어느 블럭들을 가비지 컬렉션할지를 정하는 것은 좀 더 도전적이며 여러 논문들의 주제가 되기도 했다. 최초의 LFS 논문 [RO91]의 저자들은 세그멘트를 핫(*hot*)과 콜드(*cold*)로 구분을 하는 방법을 소개하였다. 핫 세그멘트는 해당 세그멘트의 내용이 빈번하게 갱신되는 세그멘트를 말한다. 그런 세그멘트들을 위한 최선의 정책은 가비지 컬렉션하기 전에 충분히 긴 시간을 기다려서 더 많은 블럭들이 덮어 써지도록(새로운 세그멘트에) 하여, 유효 블럭의 개수를 최대한 줄이는 것이다. 그러면 유효한 블럭을 새로운 세그멘트에 복사하는 부담없이 해당 세그멘트를 해제할 수 있다. 콜드 세그멘트는 몇 개의 무효 블럭(dead block[2])들이 있지만 대부분의 블럭들은 갱신이 되지 않는 세그멘트를 일컫는다. 영화나 음악파일이 저장된 블럭들이 좋은 예이다. LFS의 저자들은 콜드 세그멘트들은 자주, 핫 세그멘트는 긴 간격으로 클리닝 하는 것이 좋다고

2) 역자 주: 최신이 아닌 블럭

결론지었고, 해당 정책으로 작동하는 기법을 개발하였다. 하지만 대부분의 정책들이 그렇듯이 완벽하지는 않다. 이후에 더 좋은 방법들이 소개되었다 [Mat+97].

46.12 크래시로부터의 복구와 로그

마지막 문제다. LFS에서는 디스크에 쓰는 도중에 시스템이 크래시되면 어떻게 되는가? 저널링을 다룬 이전 장을 돌이켜보면 갱신 중에 크래시 발생은 파일 시스템에 있어 까다로운 문제였다. LFS도 이 문제를 심도 있게 고려해야 한다.

일반적인 파일 시스템 동작에서, LFS는 쓰기 데이터를 세그멘트 버퍼에 먼저 기록하고 해당 세그멘트 버퍼를 (세그멘트가 가득차면, 또는 일정 시간이 흐르면) 디스크에 기록한다. LFS는 이러한 쓰기들을 **로그(log)**로 구성된다[3]. 즉, 체크포인트 영역에 첫 번째와 마지막 세그멘트를 가리키는 포인터를 둔다. 각 세그멘트는 다음 세그멘트를 가리키는 포인터를 둔다. 즉, 전체 세그멘트들이 일종의 링크드 리스트로 연결되는 셈이다. 체크포인트 영역의 해당 정보는 주기적으로 갱신된다. 크래시는 언제든지 발생할 수 있다(세그멘트를 쓸 때 또는 체크포인트 영역을 갱신할 때). 크래시가 발생하면 LFS는 어떻게 대처할까?

두 번째 경우, 즉 체크포인트 영역이 갱신되는 도중에 크래시가 발생하는 경우, 를 먼저 해결해 보자. 체크포인트 영역은 원자적으로 갱신되어야 한다. 체크포인트 영역이 원자적으로 갱신되는 것을 보장하기 위해 LFS는 두 개의 체크포인트 영역을 둔다. 두 개의 체크포인트 영역을 디스크의 양 끝에 위치시키고 교대로 갱신한다. 체크포인트 영역이 갱신되는 과정을 주의깊게 살펴볼 필요가 있다. LFS는 먼저 체크포인트 헤더(현재 시간 값을 포함)를 기록한 후에 체크포인트 영역에 내용을 쓰고, 그리고 최종적으로 체크포인트 영역의 마지막 블럭을 갱신한다. 마지막 블럭에도 현재 시간값이 포함되어있다. 성공적으로 체크포인트 영역이 갱신될 경우에는 헤더 블럭의 시간값이 마지막 블럭의 시간값도다 작게 된다. 체크포인트 영역 갱신 중에 크래시가 발생할 경우, 복구시에 LFS는 헤더와 마지막 영역에 저장된 시간값을 비교한다. 만약 헤더에 저장된 시간값이 더 크다면, 해당 체크포인트 영역을 갱신하는 도중 크래시가 발생한 것이다. LFS는 유효한 시간 값 쌍을 갖는 가장 최근 체크포인트 영역을 선택한다. 이를 이용하여 체크포인트 영역을 일관성 있게 갱신할 수 있다.

이제 첫 번째 문제를 다뤄보자. LFS는 약 30초마다 체크포인트 영역을 갱신한다. 따라서 디스크에 저장된 체크포인트 영역의 내용, 즉, 파일 시스템 스냅샷, 은 최신 것이 아닐수 있다. 크래시 시점을 기준으로 최대 30초 이전의 상태를 반영할 수도 있다. 재부팅 시 복구 과정에서 LFS는 체크포인트 영역을 읽어서 imap 조각들이 가리키는 곳을 확인하여 파일들과 디렉터리들을 복원한다. 이렇게 하면 마지막 수초간의 갱신 내용들이 손실되는 것을 피할 수 없다.

이 문제의 개선을 위해 LFS는 데이터베이스에서 사용되는 **롤 포워드(roll forward)** 기법을 적용하였다. 기본 개념은 최신 체크포인트 영역에서 시작한다. 복구 모듈은

3) 역자 주: 맨뒤에 계속 덧붙여 쓴다는 뜻이다.

팁: 단점을 장점으로 바꾸기

시스템에 근본적인 결함이 있을 때는 그 단점을 장점으로 전환시킬 수 있는지 살펴보아야 한다. NetApp의 WAFL은 오래된 파일 내용을 사용할 수 있도록 발전시켰다. 이전 버전을 사용할 수 있도록 하여 WAFL은 더 이상 가비지 컬렉션에 대해서 고려하지 않아도 되었다. 한 번 멋지게 비틀어서 훌륭한 기능으로 만들었을 뿐만 아니라 LFS의 가비지 컬렉션 문제도 제거하였다. 이와 유사한 다른 예제를 시스템에서 찾아 볼 수 있을까? 의심의 여지가 없다 하지만 스스로 한 번 생각해 봐야 할 것이다. 왜냐하면 이 장은 이제 대문자 "O"로 끝이 났기 때문이다. 끝. 다 끝났다. 결판났다. 이제 간다. 평안하시길!

체크포인트 영역을 읽어들여 세그멘트 리스트의 마지막 세그멘트 위치를 파악한다. 각 세그멘트는 다음 세그멘트를 가리키는 포인터를 가지고 있다. 체크포인트가 가리키는 마지막 세그멘트를 읽어서, 이 세그멘트가 가리키는 다음 세그멘트의 존재 여부를 검사하고, 필요하면 체크포인트의 "마지막 세그멘트"를 가리키는 포인터를 갱신한다. 이런식으로 계속 다음 세그멘트를 찾아나간다. 세그멘트 리스트를 순회하여, 마지막 체크포인트 이후의 데이터와 메타데이터를 복구한다. 더 자세한 내용은 우수학위논문상을 수상한 Rosenblum의 학위 논문을 참고하자 [Ros92].

46.13 요약

LFS는 새로운 디스크 갱신 방법을 소개하였다. 파일들을 원래의 자리에 덮어쓰는 대신, LFS는 항상 디스크에서 사용되지 않은 부분에 쓰고 나중에 오래된 것들을 가비지 컬렉션해서 공간을 확보한다. 이 방법을 데이터베이스 시스템에서는 **쉐도우 페이징**(**shadow paging**)이라고 부르며 [Lor77], 파일 시스템에서는 때로 **쓰기 시 복사**(**copy-on-write**)라고 부른다. LFS는 모든 갱신들을 메모리 상의 세그멘트에서 다같이 모아서 순차적으로 쓸 수 있기 때문에 이 기법의 쓰기는 매우 효율적이다.

단점이 있다면 LFS 기법은 가비지를 만들어 낸다는 것이다. 오래된 데이터 블럭들이 디스크 여기저기에 흩어져있다. 공간을 회수하려면 오래된 세그멘트를 주기적으로 가비지 컬렉션해야 한다. LFS의 가비지 컬렉션이라는 부분이 논란의 핵심이 되었으며 가비지 컬렉션 비용 [Sel+95]에 대한 우려때문에 LFS는 그다지 큰 여파를 일으키지는 못했다. 하지만 NetApp의 **WAFL** [HLM94], Sun의 **ZFS** [BM07] 그리고 Linux의 **btrfs** [Mas07]와 같은 상용 파일 시스템은 LFS와 유사한 Copy-on-write 방식으로 디스크에 기록한다. LFS의 철학은 현대 파일 시스템에 존재한다. WAFL과 같은 경우를 보면 가비지 컬렉션 문제를 역이용하여 새로운 기능으로 발전시켰다. 파일 시스템의 이전 버전들을 **스냅샷**(**snapshot**)으로 유지하여 사용자가 실수로 현재 것을 지웠다하더라도 오래된 파일들을 접근할 수 있도록 하였다.

참고 문헌

[BM07] **"ZFS: The Last Word in File Systems"**
Jeff Bonwick and Bill Moore
URL: `%7Bhttp://opensolaris.org/os/community/zfs/docs/zfs%20last.pdf%7D`
ZFS에 대한 발표 자료이다. 불행하게도 ZFS를 설명하는 좋은 논문이 없다.

[HLM94] **"File System Design for an NFS File Server Appliance"**
Dave Hitz, James Lau, and Michael Malcolm
USENIX Spring '94
수십억 달러의 저장 장치 기업인 NetApp은 고속의 NFS 장비를 위해 WAFL을 만들면서 LFS와 RAID의 많은 개념들을 적용하였다.

[Lor77] **"Physical Integrity in a Large Segmented Database"**
R. Lorie
ACM Transactions on Databases, 1977, Volume 2:1, pages 91-104
쉐도우 페이징의 원개념이 소개되었다.

[Mas07] **"The Btrfs Filesystem"**
Chris Mason
URL: `oss.oracle.com/projects/btrfs/dist/documentation/btrfs-ukuug.pdf`
최신 쓰기 시 복사(copy-on-write) Linux 파일 시스템으로 그 중요도와 사용자가 서서히 늘어나고 있다.

[Mat+97] **"Improving the Performance of Log-structured File Systems with Adaptive Methods"**
Jeanna Neefe Matthews, Drew Roselli, Adam M. Costello, Randolph Y.Wang, and Thomas E. Anderson
SOSP 1997, pages 238-251, October, Saint Malo, France
좀 더 나은 LFS의 가비지 컬렉션 정책들을 다룬 좀 더 최신의 논문이다.

[McK+84a] **"A Fast File System for Unix"**
Marshall K. McKusick, William N. Joy, Sam J. Leffler, and Robert S. Fabry
ACM TOCS, August, 1984, Volume 2, Number 3
원조 FFS 논문이다. 상세 내용은 FFS를 다룬 장을 참고하자.

[McK+84b] **"A Fast File System for Unix"**
Marshall K. McKusick, William N. Joy, Sam J. Leffler, and Robert S. Fabry
ACM Transactions on Computing Systems.
FFS에 대해서 충분히 알고 있다. 그렇지 않은가? 그렇지만 이와 같은 참고 문헌은 책에서 여러 번 소개해도 괜찮다.

[Mog94] **"A Better Update Policy"**
Jeffrey C. Mogul
USENIX ATC '94, June 1994
이 논문에서 Mogul은 읽기 워크로드를 너무 오랫동안 버퍼에 두었다가 한 번에 큰 청크로 디스크에 쓰게 되면 피해를 입을 수 있다는 것을 밝혔다. 그러므로 그는 쓰기를 좀 더 자주 그리고 작은 묶음으로 보내기를 추천한다.

[Pat98] **"Hardware Technology Trends and Database Opportunities"**
David A. Patterson
ACM SIGMOD '98 Keynote Address, Presented June 3, 1998, Seattle, Washington
URL: `http://www.cs.berkeley.edu/.pattrsn/talks/keynote.html`
컴퓨터 시스템의 기술 동향을 다룬 대단한 발표 자료이다. Patternson이 조만간에 이런 자료를 또 만들기를 기대해 본다.

[Ros92] **"Design and Implementation of the Log-structured File System"**
Mendel Rosenblum
URL: `http://www.eecs.berkeley.edu/Pubs/TechRpts/1992/CSD-92-696.pdf`
논문에서 소개하지 못한 LFS의 여러 세부 사항을 설명한 상 받은 학위 논문이다.

[RO91] **"Design and Implementation of the Log-structured File System"**
Mendel Rosenblum and John Ousterhout
SOSP '91, Pacific Grove, CA, October 1991
SOSP에서 처음으로 소개된 LFS 논문이다. 수백 편의 다른 논문들에 의해 참조되었고 여러 실제 시스템들의 영감이 되었다.

[Sel+95] **"File system logging versus clustering: a performance comparison"**
Margo Seltzer, Keith A. Smith, Hari Balakrishnan, Jacqueline Chang, Sara McMains, and Venkata Padmanabhan
USENIX 1995 Technical Conference, New Orleans, Louisiana, 1995
때로는 LFS의 성능이 문제가 있다는 것을 보여준 논문이다. 특히 `fsync()`(데이터베이스 워크로드임)를 많이 사용하는 워크로드에서 안 좋다. 그 당시 이 주제에 관한 논란이 많았다.

[SO90] **"Write-Only Disk Caches"**
Jon A. Solworth and Cyril U. Orji
SIGMOD '90, Atlantic City, New Jersey, May 1990
쓰기 버퍼링에 대한 초기 연구로 그것에 대한 장점을 다루었다. 하지만, 너무 오랫동안 버퍼링하면 손해가 있을 수 있다. 자세한 내용은 Mogul[Mog94]를 참고하자.

[Zha+12] **"De-indirection for Flash-based SSDs with Nameless Writes"**
Yiying Zhang, Leo Prasath Arulraj, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau
FAST '13, San Jose, California, February 2013
플래시 기반의 저장 장치를 만들기 위한 새로운 방법을 다룬 우리의 논문이다. FTL(플래시 변환 계층)은 대체적으로 로그 기반으로 만들어졌기 때문에 플래시 기반 장치들에서도 LFS에서 보았던 동일한 문제들이 나타난다. 그 문제는 LFS가 imap을 사용하여 깔끔하게 해결한 재귀적 갱신 문제이다. imap과 유사한 구조가 대부분의 SSD들에도 존재한다.