[REPORT]

■과 목 명: 운영체제 (SWE3004-42)

■담 당 교 수: 신동군 교수님

■제 출 일: 2022년 6월 4일

■학　　과: 수학과

■학　　번: 2016314786

■성　　명: 김호진

Chap 43. Log-structured File Systems

수학과 김호진 (2016314786)

1990년대 초, 버클리 대학의 John Ousterhout 교수와 대학원생 Mendel Rosenblum은 다음의 문제들을 해결하기 위해서 Log-structured File System으로 알려진 새로운 파일 시스템을 개발했다.

1. 시스템 메모리 증가: 메모리가 커지면서 더 많은 데이터를 메모리에 캐시 할 수 있게 되었다. 이에 따라 디스크의 읽기 작업이 메모리의 캐시에 의해 서비스되었고 디스크에서 쓰기 작업의 비중이 높아지는 원인이 되었다. 그 결과, 쓰기 작업에 의해 파일 시스템의 성능이 결정되었다.

2. 랜덤 I/O 성능과 순차 I/O 성능 사이에 큰 차이가 발생했다: 하드 디스크의 성능이 좋아지면서 대역폭은 증가했지만 Seek time과 Rotate time 비용은 크게 줄지 않았다. 그 대신 디스크를 연속적으로 사용하여 Seek time과 Rotate time을 줄였고 상당한 성능상의 이점을 얻었다.

3. 기존의 파일 시스템은 대다수의 일반적인 워크로드에서 성능이 저하된다: FFS는 한 블록의 파일을 만들기 위해 많은 쓰기 작업을 수행한다. 이러한 경우, 성능을 좋게 하기 위해서 모든 블록을 동일한 블록 그룹 내에 배치하더라도 많은 short seek과 그에 따른 rotational delay가 유발되기 때문에 성능이 peak sequential bandwidth에 훨씬 미치지 못한다.

4. 파일 시스템은 RAID를 인식하지 못한다: RAID-4와 RAID-5는 단일 블록에 대한 쓰기와 같이 작은 쓰기 작업에 대해 4개의 I/O가 발생하는 문제를 가진다. 기존의 파일 시스템은 이렇게 비효율적인 RAID 쓰기 작업을 효율적으로 수행할 방법이 없었다.

따라서 이상적인 파일 시스템은 쓰기 성능에 초점을 맞추고 디스크의 순차적 대역폭을 사용하고자 한다. 또한, 디스크 상의 메타데이터 구조를 자주 업데이트하는 일반적인 워크로드에서도 우수한 성능을 발휘하며 RAID에서도 효율적으로 작동한다. Rosenblum과 Ousterhout가 새롭게 도입한 LFS는 디스크에 뭔가를 쓸 때 모든 업데이트를 메모리의 세그먼트에 버퍼링한다. 이때 기존 데이터를 덮어쓰지 않고 항상 세그먼트의 빈 공간에 쓰며, 세그먼트가 크기 때문에 디스크가 효율적으로 사용되고 파일 시스템의 성능을 좋게 만든다.

[Writing To Disk Sequentially]

모든 업데이트를 디스크에 파일 시스템 상태로 순차적으로 쓸 수 있게 변환하는 과정을 알아보기 위해서 데이터 블록 D를 파일에 쓰는 간단한 예를 살펴보았다. 사용자가 데이터 블록을 쓸 때 디스크에 기록되는 것은 데이터만 아니라 업데이트를 해야 하는 다른 메타데이터도 있기 때문에 디스크 블록 D를 가리키는 inode도 함께 써줘야 한다. 즉, LFS의 핵심 전략 중 하나는 데이터 블록과 inode 등 파일 업데이트에 필요한 정보들을 디스크에 연속적으로 쓰는 것이다.

[Writing Sequentially And Effectively]

그러나 디스크에 연속적으로 쓰는 것만으로는 효율적인 쓰기를 보장할 수 없다. 예를 들어, 시간 T에서 주소 A에 단일 블록을 쓰고, 잠시 기다린 이후 주소 A+1에 무엇인가를 쓰고자 한다면 디스크의 회전으로 인한 Rotation time이 발생하게 된다. 따라서 우수한 쓰기 성능을 얻기 위해서는 어느 정도의 쓰기 작업을 모은 뒤 이를 한 번에 처리해야 한다. 이를 수행하기 위해서 LFS는 write buffering이라는 기술을 사용한다. LFS는 디스크에 쓰기 동작을 수행할 때 메모리 내 세그먼트에서 업데이트를 버퍼링하고, 이후 전체 세그먼트들을 디스크에 한 번에 쓴다. 세그먼트의 크기가 충분히 클수록 이러한 동작은 효율적이다.

[How Much To Buffer?]

그렇다면 디스크에 쓰기 전 LFS는 얼마나 많은 업데이트를 버퍼링 해야 할까? 물론 이는 디스크의 성능에 따라 다르겠지만, 다음 예시를 통해 대략적으로 나타내 보았다.

쓰기 작업을 수행하기 위해서 디스크의 주소를 찾아가는데 (rotation and seek overhead) 대략 T_P초가 소요되고 디스크 전송 속도가 R_{peak} MB/s라고 가정하자. LFS에서는 Positioning cost라는 고정 비용을 지불하기 때문에 Segment의 크기가 커질수록 성능은 좋아지고 최대 대역폭 달성에 가까워질 것이다.

보다 더 구체적인 답을 얻기 위해, 우리가 D MB 용량의 데이터를 디스크에 쓴다고 가정해보자. 이 데이터를 쓰는 시간(T_{write})은 위치 결정 시간($T_{position}$)에 D를 전송하는 시간(D/R_{peak})을 더하여 결정된다. 따라서 쓰기 작업의 데이터 양을 총 쓰기 시간으로 나눈 값인 유효 쓰기 속도($R_{Effective}$)는 다음과 같다.

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

이러한 유효율을 최고에 가깝게 하고자 하기 때문에($R_{effective} = F * R_{peak}$, $0 < F < 1$), 이를 이용하여 기존의 식으로부터 새로운 관계식을 얻을 수 있다.

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

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

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

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

[Problem: Finding Inodes]

FFS나 UNIX File System의 경우, inode를 디스크의 특정 위치에 배열 형태로 위치시키기 때문에 쉽게 찾을 수 있었다. 그러나 LFS에서 inode들은 디스크 전체에 흩어져 있고, 기존의 inode를 덮어쓰지 않기 위해 최신 버전의 inode가 계속 움직이면서 기존의 방식으로는 inode를 찾을 수 없게 되었다.

[Solution Through Indirection: The Inode Map]

이러한 문제를 해결하기 위해, LFS 설계자들은 inode map(imap)이라 불리는 데이터 구조를 사용하여 inode number와 inode 사이의 간접적인 레벨을 도입했다. imap은 inode number를 입력으로 받아서 최신 inode의 디스크 주소를 생성하고, inode가 디스크에 기록될 때마다 inode의 새로운 위치를 업데이트 한다.

이러한 imap은 디스크에 영구히 유지되어야 하는데, 기존처럼 디스크의 특정 부분에 위치시킨다면 inode를 업데이트 할 때 마다 imap에 쓰기를 해야 하므로 성능이 저하될 수밖에 없다. 따라서 LFS에서는 inode가 새로 쓰여지는 위치 바로 옆에 imap을 배치한다. 이를 정리하자면 LFS는 파일에 블록을 추가할 때마다 디스크에 새로운 데이터 블록, inode, inode map의 일부를 함께 기록한다는 것이다.

[Completing The Solution: The Checkpoint Region]

그런데 위 방식을 사용하면 Inode map의 일부가 디스크에 흩어지게 된다. 그래서 LFS는 디스크의 특정 부분에 Checkpoint Region(CR)이라는 inode map을 찾기 위한 데이터를 배치한다. CR은 30초의 긴 주기로 업데이트 되기 때문에 성능에 큰 영향을 주지 않는다. 즉, on-disk layout의 전체 구조는 CR을 포함하며, inode map 조각들은 각 indoe의 주소를 포함하고 inode는 파일 및 디렉토리를 가리킨다.

[Reading A File From Disk: A Recap]

LFS의 작동 방식을 확실히 이해하기 위해서 디스크에서 파일을 읽기 위한 과정을 살펴봤을 때, 시작할 만한 memory가 없다면 읽어야 할 첫 번째 on-disk 데이터 구조는 Checkpoint Region이다. CR은 전체 inode map에 대한 포인터를 포함하기 때문에 LFS는 전체 inode map을 읽고 메모리에 캐시한다. 이후, 파일의 inode number가 주어지면 LFS는 imap 상에서 inode number와 대응되는 inode disk address를 찾아보고 inode의 최신 버전을 읽는다. 이 시점에서 LFS는 전형적인 UNIX File System과 동일하게 파일로부터 블록을 읽기 위해 필요에 따라 direct pointer나 indirect pointer 또는 doubly-indirect pointer를 사용한다. 일반적인 경우, LFS는 디스크에서 파일을 읽을 때 일반적인 파일 시스템과 동일한 수의 I/O를 수행한다.

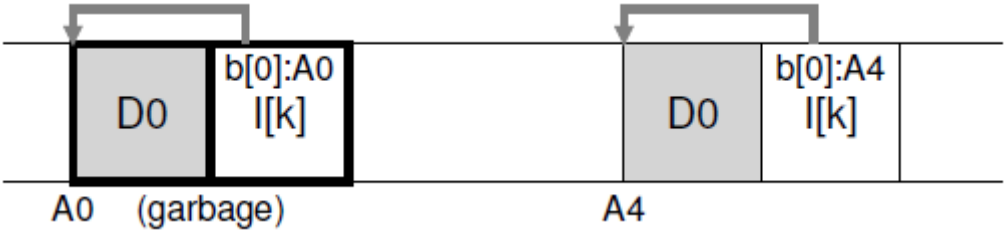
[What About Directories?]

지금까지는 inode와 data block에 대해서만 고려하였다. 그러나 파일 시스템의 파일에 접근하기 위해서는 우선적으로 디렉터리에 접근해야 한다. 다행히도 디렉터리 구조는 기본적으로 (이름, inode number)의 집합이라는 점에서 UNIX File System과 동일하다. 그러므로 LFS는 디스크에 파일을 만들 때 새로운 inode와 일부 데이터에 더하여 해당 파일을 참조하는 디렉토리 데이터와 inode도 함께 쓴다.

그런데 데이터가 업데이트 될 때마다 디렉토리의 데이터를 매번 변경하면 디렉토리 트리 구조를 계속해서 변경해야 하기 때문에 하나의 파일 수정으로 인해 파일 시스템 전체 구조가 바뀌게 되는 문제가 발생한다. LFS는 이러한 문제를 inode map을 통해 해결하였다. inode의 위치가 변경되더라도 변경사항은 디렉터리에 바로 반영되지 않으며, 디렉터리가 동일한 (이름, inode number) mapping을 가지고 있는 동안 inode map만 업데이트 된다. 이처럼 LFS는 indirection을 이용하여 recursive update problem을 피했다.

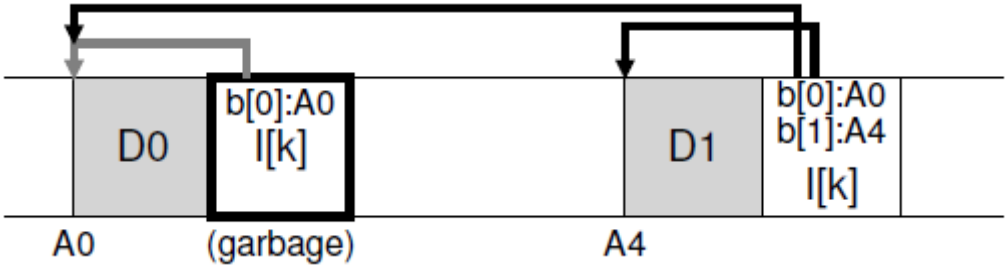
[A New Problem: Garbage Collection]

LFS에서는 또 다른 문제가 존재한다. LFS는 반복적으로 파일의 최신 버전을 디스크의 새 위치에 쓴다. 이러한 프로세스는 쓰기 작업을 효율적으로 유지하지만, 디스크 전체에 이전 버전의 파일 구조를 남기게 된다. 이렇게 남겨진 오래된 버전의 파일을 garbage라고 부른다. 예를 들어, 단일 데이터 블록 D0을 가리키는 inode number k로 참조되는 기존 파일이 있다고 가정해 보자. 해당 블록을 업데이트하면 새로운 inode와 데이터 블록이 생성되며, 이때 디스크 상의 LFS 레이아웃은 다음과 같다.



다이어그램에서 확인할 수 있듯이 inode와 데이터 블록 모두 오래된 버전(왼쪽)과 현재 버전(오른쪽)이 디스크에 존재한다. 데이터 블록을 업데이트하는 간단한 동작 속에서 수많은 새로운 구조들이 LFS에 의해 유지되어야 하며, 이를 위해서 디스크에 이전 버전의 블록들을 남겨두어야 한다.

이번에는 원본 파일 k에 블록을 추가한다고 가정해 보자. 새로운 버전의 inode가 생성되기는 하지만 inode는 여전히 이전 데이터 블록을 가리킨다. 따라서 이 파일은 여전히 live하다.



이러한 방법은 이전 버전을 유지하면서 사용자가 오래된 파일 버전을 복원할 수 있도록 만들고, 파일 시스템이 파일의 다른 버전을 추적하기 때문에 versioning file system으로 알려져 있다. 그러나 LFS는 최신 live 버전의 파일만을 고려하므로 오래된 dead version의 파일을 주기적으로 찾아서 정리해야 한다.

Cleaning process는 프로그래밍 언어에서 발생하는 일종의 garbage collection으로, 프로그램에서 사용되지 않은 메모리를 자동으로 비우는 기술이다. 만약 LFS cleaner가 cleaning 중에 단일 데이터 블록, inode 등을 간단히 검사만 하고 해제한다면, 디스크에서 할당된 공간 사이에 일부 빈 구멍이 혼재된 파일 시스템이 생성된다. 이로 인해 LFS가 디스크에 순차적으로 쓸 수 있는 대규모 인접 영역을 찾지 못하게 되고 쓰기 성능이 상당히 저하된다. 그렇기 때문에 LFS cleaner는 세그먼트 단위로 작동한 이후 쓰기를 위해 필요한 큰 공간을 정리한다. LFS cleaner는 정기적으로 오래된 세그먼트를 읽고 이러한 세그먼트 내에 어떤 블록이 존재하는지 확인한 뒤 live 블록만을 포함된 새로운 세그먼트를 작성하여 오래된 세그먼트를 사용할 수 있게 한다.

그러나 아직도 두 가지 문제점이 존재한다. 첫 번째는 LFS가 세그먼트 내에서 어떤 블록이 활성 상태이고 어떤 블록이 비활성 상태인지 구분하는 메커니즘에 관한 문제이며, 두 번째는 cleaner가 얼마나 자주 가동되어야 하고, 어떤 세그먼트를 선택해야 할지 결정하는 정책에 관한 문제이다.

[Determining Block Liveness]

먼저 메커니즘에 대한 문제를 생각해보자. 디스크 상의 세그먼트 S 내에서 데이터 블록 D가 주어졌을 때, LFS는 D가 live인지 아닌지를 판단할 수 있어야 한다. 이를 위해서 LFS는 세그먼트 헤더에 존재하는 segment summary block에 inode number와 offset에 대한 정보를 가진다. 만약 inode number와 offset을 통해 가리키는 디스크 주소가 정확하다면, 블록 D를 활성 상태로 판단한다.

LFS에서는 liveness를 결정하는 프로세스를 효율적으로 만들기 위해서 몇 가지 shortcut을 사용한다. 만약 파일이 잘리거나 삭제되면 LFS는 해당 version number를 증가시킨 뒤 imap에 새로운 version number를 기록한다. 이렇게 LFS는 디스크 상의 세그먼트에 version number를 기록함으로써 추가적인 읽기 작업을 피한다.

[A Policy Question: Which Blocks To Clean, And When?]

이 밖에도 LFS는 어떤 주기로 어느 블록을 cleaning 해야 할 지 결정하기 위한 정책을 가져야 한다. 먼저 블록을 정리하기 위한 시간은 idle time 동안 주기적으로 진행하거나 디스크가 가득 찼을 경우에 진행하면 된다. 어떤 블록을 정리할지 결정하는 것은 이보다 더 어려운 문제이며, 많은 연구 논문의 주제가 되어 왔다. 기존의 LFS 논문의 경우, hot 세그먼트와 cold 세그먼트를 분리하는 방식을 사용한다. hot 세그먼트는 컨텐츠를 자주 덮어쓰는 세그먼트로 많은 블록을 모아서 한 번에 처리하는 게 효율적이다. 그와 반대로 cold 세그먼트는 dead 블록이 몇 개 있을 수 있지만 나머지 내용은 비교적 안정적이기 때문에 먼저 처리하는 것이 좋다. 그러나 다른 대부분의 정책들과 마찬가지로 이 정책 역시 완벽한 방안은 아니다.

[Crash Recovery And The Log]

마지막으로 데이터를 쓰는 동안 시스템에 문제가 발생했을 때 데이터를 복구하는 방안에 대해 살펴보자. LFS가 정상적으로 작동한다면, 디스크에 세그먼트를 쓰고 이를 log로 구성할 것이다. 이때 Checkpoint Region은 head와 tail 세그먼트를 가리키고, 각 세그먼트들은 다음 세그먼트를 가리킨다. LFS는 정기적으로 Checkpoint Region을 업데이트하며 클리너는 주기적으로 garbage collection을 수행한다. 그런데 세그먼트나 Checkpoint에 쓰기 작업을 수행하는 도중에 충돌이 발생할 수 있다.

Checkpoint Region을 쓸 때 발생하는 충돌을 해결하는 방법을 먼저 살펴보자. 실제 LFS는 CR 업데이트가 원자적으로 수행되도록 하기 위해서 디스크 양 끝에 CR을 하나씩 보관하고 번갈아 기록한다. 또한, LFS는 CR을 업데이트 할 때마다 헤더, CR 본문, 마지막 블록 등을 기록하는데 마지막 블록에 timestamp 정보를 남긴다. CR 업데이트 중 시스템이 충돌하는 경우, LFS는 헤더와 마지막 블록의 timestamp를 확인하여 일치하지 않으면 쓰기 작업을 다시 수행한다. 그 결과, LFS는 Checkpoint Region에서 일관된 업데이트가 가능해진다.

다음으로 세그먼트를 쓸 때 발생하는 충돌을 해결하는 방법을 살펴보자. LFS는 Checkpoint Region을 30초에 한 번 정도의 주기로 업데이트하기 때문에 파일 시스템에 존재하는 정보가 오래된 정보일 수 있다. 이를 방지하기 위해서 LFS는 마지막 Checkpoint Region로부터 시작하여 log의 끝을 찾은 뒤 다음 세그먼트를 읽는다. 이후 해당 영역에 유효한 업데이트가 존재하는지 확인하고 필요하다면 업데이트를 해준다. 이러한 방법을 통해서 마지막 체크포인트 이후에 작성된 데이터 및 메타데이터의 상당 부분을 복구할 수 있게 되었다.

[Summary]

LFS는 디스크를 업데이트하는 새로운 방식을 도입했다. LFS에서는 파일을 덮어쓰는 대신 디스크의 사용되지 않는 부분에 새로운 데이터를 쓰고 이전 데이터는 garbage collection을 통해 처리한다. LFS는 모든 업데이트를 in-memory segment로 수집하며 순차적으로 쓰기를 수행하기 때문에 좋은 성능을 가진다. 특히 LFS가 제공하는 대용량 쓰기는 다양한 디바이스에서 탁월한 성능을 발휘한다. 대용량 쓰기를 통해 하드 드라이브의 포지셔닝 시간을 최소화할 수 있으며, RAID-4 및 RAID-5와 같은 parity 기반 RAID에서의 작은 쓰기 문제를 완전히 방지할 수 있다. 또한, Flash based SSD에서 고성능을 발휘하기 위해서 대용량 I/O가 필요하다는 연구가 존재하는데 이를 위해 LFS가 활용될 수 있다.

[My opinion]

이번 과제를 통해서 Log-structured File Systems(LFS)에 대한 전반적인 내용을 배울 수 있었다. LFS를 개발하게 된 원인으로부터 시작하여 LFS의 기본 구조와 동작 원리에 대해 알아보았고, LFS가 Garbage collection, Determining Block Liveness Mechanism, Determining Block to Clean Policy, Crash Recovery 등의 문제를 해결하기 위해 사용한 방안들을 차례대로 확인하였다. 최종적으로 LFS가 제공하는 장점에 대해 알아보면서 하드 드라이브, RAID, Flash based SSD 등과 같은 다양한 디바이스에서 LFS가 활용될 수 있음을 알게 되었다.

앞서 언급했듯이 LFS는 업데이트된 데이터를 메모리에 충분히 모은 뒤 이 데이터를 한 번에 순차 쓰기로 디스크에 기록함으로써 높은 쓰기 성능을 실현한 파일 시스템 구조이다. 그런데 실제 시스템의 경우, 디스크와 메모리 상의 일관성을 위해서 동기화가 발생하기 때문에 LFS가 메모리에 데이터를 충분히 모으지 못한 채 쓰기 동작이 수행되는 모습을 보인다고 한다. 쓰기 동작이 자주 발생하게 되면, 클리너의 오버헤드를 증가시키고 더 많은 메타 데이터를 기록하게 되면서 많은 문제가 나타나게 된다. 그렇기 때문에 이 문제를 해결하기 위한 많은 연구들이 진행되어 왔으며 비휘발성 메모리를 이용해서 동기화를 없애고 작은 단위의 쓰기를 효과적으로 활용하도록 LFS와 운영체제의 관련된 서브 시스템들을 변경하는 방안 등이 대안으로 제시되었다. 본 과제를 수행하면서 LFS의 기본적인 동작 원리를 넘어 LFS가 가지는 다양한 장점들과 문제점들을 알 수 있었고 개인적으로도 매우 유익한 경험이 되었다.