DOI: 10.3745/KIPSTA.2009.16-A.2.101

파일 시스템 노화를 해소하기 위한 자동적인 단편화 해결 시스템의 설계와 구현

이 준 석[†] · 박 현 찬[††] · 유 혁[†††]

요 약

파일 시스템의 단편화 현상을 해결하기 위한 기존 기법들은 디스크 조각 모음과 같이 특정 시점에 집중된 디스크의 연산이 요구된다. 본 논문에서는 이러한 단점을 해소하기 위해 디스크의 연산 처리를 분산시키는 자동적이고, 지속적인 단편화 해소 시스템을 설계하고 구현하고자 한다. 이를 위해 우리는 단편화 측정을 위한 자동적인 레이아웃 스코어링(ALS: Autonomic Layout Scoring)기법과 디스크의 연산 처리를 분산시키기 위한 기법으로 디스크의 유휴 시간(idle time)을 찾아 복사를 수행하는 지연 복사(Lazy-copy)기법을 제안한다. 두 기법은 우선 자동적인 레이아웃 스코어링을 통해 단편화 된 대상 파일을 검색하고, 검색된 파일을 옮길 수 있는 연속적인 빈 공간이 있을 경우, 파일의 유실을 막기 위해 지연 복사를 수행한 후 아이노드의 정보를 수정함으로써 단편화 현상을 해결한다. 본 논문에서 제시한 시스템을 실제 리눅스(linux) 환경에 적용하여 단편화 된 작은 파일의 레이아웃 스코어링을 측정 한 결과 기존 EXT2 파일 시스템보다 2.4%~10.4% 정도의 레이아웃 스코어링이 향상된 것을 볼 수 있었으며, 실험 디스크에 파일 크기에 따른 읽기/쓰기를 실행하여 성능을 측정한 결과에서도 EXT2 파일 시스템과 비교하여 쓰기성능에서는 1%~8.5%, 읽기 성능에서는 1.2%~7.5% 정도의 향상된 결과를 보았다. 이 시스템을 이용하면 수동적인 관리 없이도 자동적으로 사용자의 I/O 작업에 대한 방해 없이 단편화 현상을 지속적으로 해소할 수 있다.

키워드 : 파일 시스템, 조각모음

Design and Implementation of Autonomic De-fragmentation for File System Aging

Junseok Lee[†] · Hyunchan Park[††] · Chuck Yoo[†††]

ABSTRACT

Existing techniques for defragmentation of the file system need intensive disk operation for some periods at specific time such as disk defragmentation program. In this paper, for solving this problem, we design and implement the automatic and continuous defragmentation free system by distributing the disk operation. We propose the Automatic Layout Scoring(ALS) mechanism for measuring defragmentation degree and suggest the Lazy Copy mechanism that copies the defragmented data at idle time for scattering the disk operation. We search the defragmented file by Automatic Layout Scoring mechanism and then find for empty spaces for that searched file. After lazy copy of searched fils to empty space for preventing that file from being lost, the algorithm solves the defragmentation problem by updating the I-node of that file. We implement these algorithms in Linux and evaluate them for small and defragmented file to get the layout scoring. We outperform the Linux EXT2 file system by 2.4%~10.4% in layout scoring evaluation. And the performance of read and write for various file size is better than the EXT2 by 1%~8.5% for write performance and by 1.2%~7.5% for read performance. We suggest this system for solving the problem of defragmentation automatically without disturbing the I/O task and manual management.

Keywords : File System, Defragmentation

1. 서 론

1.1 연구배경

이차적인 저장장치로 널리 사용되고 있는 하드디스크는 기술의 발달로 인해 점점 대형화 되고, 가격 면에서도 사용자의 부담을 많이 줄여주었다. 이로 인해 하드디스크에 저장해두고 사용하는 파일의 수와 양도 증가하고 있다. 이런 현상은 파일의 끊임없는 삭제와 생성으로 연결되었고, 파일 시스템의 단편화(fragmentation) 현상은 심화 되었다. 기술의 발전으로 인해 하드디스크의 데이터 전송률은 크게 향상되었지만, 하드디스크의 특성상 기계적인 회전과 헤드(head)

※ 본 연구는 BK21 2단계 사업의 지원과 서울시 산학연 협력사업의 지원으로 수행된 연구임.
† 준 회 원 : 육군3사관학교 교수부 컴퓨터공학과
†† 준 회 원 : 고려대학교 컴퓨터학과 석·박사통합과정
††† 종신회원 : 고려대학교 컴퓨터공학과 교수
논문접수 : 2008년 11월 18일
수 정 일 : 1차 2009년 2월 13일
심사완료 : 2009년 2월 13일

의 움직임을 통해 파일의 I/O를 처리해야하는 한계는 전체 시스템의 병목현상(deadlock)을 유발하는 주원인이 되며, 파일 시스템의 성능 저하는 시스템의 전체 성능에 큰 영향을 끼치게 된다[1, 2].

파일 시스템의 단편화 현상은 파일 시스템의 노화(file system aging)으로 인해 발생하는 가장 큰 문제점으로써, 반복되는 파일의 삭제와 생성, 그리고 여러 프로세스가 동시에 디스크 블록을 할당 받으려 할 때에 발생한다. 파일 시스템의 단편화 현상을 잘 해결하지 못하게 되면, 파일 시스템 고유의 블록 할당 정책(block allocation policy)[3]를 효율적으로 수행 할 수 없게 되고, 디스크의 블록 배치 레이아웃(disk block allocation layout)[4, 5]을 유지할 수 없다. 이런 이유로 인해 파일 시스템의 성능은 저하되고, 대부분의 경우에는 스스로 회복하기 어려워진다. 파일 시스템 노화로 인한 단편화 현상 문제는 파일 시스템의 초기 디자인에서부터 다루어지는 중요한 고려사항이며, 많은 연구가 이루어지고 있다. 일반적으로 단편화 현상에 대한 해결 방안은 예방 및 최소화가 아니라, 단편화 현상 발생 이후의 해결에 초점을 두고 있다. 예를 들어 이 문제를 해결하기 위한 많은 상업적 어플리케이션들을 볼 수 있다. 가장 대표적으로 마이크로소프트의 윈도우즈 운영체제 시리즈에 포함되어 판매되는 "디스크 조각모음" 프로그램이 그러하다. 현재 디스크에 저장되어있는 파일들의 단편화를 제거해주는 프로그램이며, 대부분의 어플리케이션들은 비슷한 방식으로 접근하고 있다. 파일 시스템의 초기 디자인 단계에서 고려하는 경우는 리눅스의 FFS(Fast File System)을 개량한 것[4]을 예로 들 수 있다.

그러나 기존 접근 방법들은 다음과 같은 문제점들을 지니고 있다. 우선 어플리케이션을 통해 해결하는 방법은 운영체제 커널의 파일 시스템과 무관하게 동작하기 때문에 비효율적이고, 대단히 많은 양의 I/O를 발생시키기 때문에 다른 작업을 정상적으로 수행할 수 없을 정도로 시스템 자원을 소모한다. FFS[6, 7]에서 사용된 방식은 기존 정책을 개량하여 성능을 높이는데 성공하기는 했지만, 이미 발생한 단편화 현상을 해소할 수는 없다. 따라서 이러한 문제점을 해결할 수 있는 새로운 방안이 필요하며, 그것은 자동적이고, 지속적인 문제 해결 방안이 될 수 있어야 한다.

1.2 연구목표

본 논문에서는 리눅스 파일 시스템으로 널리 사용되고 있는 EXT2 파일 시스템을 대상으로 자동적이고 지속적으로 단편화를 측정할 수 있는 시스템과 측정된 단편화의 정도를 바탕으로 단편화를 해소할 수 있는 시스템을 설계하고 구현하였다. 본 논문에서 제안하는 단편화 측정 시스템은 ALS (Autonomic layout-scoring)로 단편화를 해결하는 시스템을 LCD-FS(Lazy-copy Defragmentation File System) 이라 각각 명명하였다. 설계에 따른 시스템 구현은 리눅스 2.6.18 버전에서 모듈화 된 EXT2 파일 시스템을 수정하여 이루어졌다.

ALS를 통해 작은 파일(12블록 이하)을 대상으로 단편화 된 파일을 찾고, 대상파일의 아이노드 정보를 이용하여 파일 크기를 알아내고, 단편화 된 파일이 새로운 공간에 할당되기 위해 파일 크기에 맞는 새로운 연속된 빈 공간을 찾아 예약할당한 후 I/O 스케줄러에게 I/O를 요청하여 실제 복사가 이루어질 수 있도록 한다. 실제 파일의 복사가 이루어진 후에는 아이노드 정보의 수정을 통해 단편화가 해소되는 시스템을 설계하고 구현하는 것을 연구 목표로 하고 있다. 이와 더불어 실험을 통해 EXT2 파일 시스템과 LCD-FS의 성능을 비교하였고, LCD-FS의 사용으로 인한 오버헤드의 문제점이 발생하는지의 여부도 측정한다.

1.3 논문의 구성

본 논문에서는 자동적인 단편화 해소 시스템을 제시하기 위해 ALS와 LCD-FS을 설계하고 구현 하였다. 전체적인 논문의 구성은 다음과 같다.

2장은 관련 연구로 FFS(Fast File System)의 디스크 할당 정책에 대하여 소개와, 기존에 사용되고 있는 레이아웃 스코어링 기법에 대한 방법을 기술하고, 이 기법의 한계점에 대해 분석하였다. 또한, De-fragmentation File System의 연구에서는 버퍼캐시(buffer cache)를 이용한 단편화 해결방법의 소개와 함께 문제점을 제시하였다. 3장에서는 본 논문에서 제시한 ALS와 LCD-FS의 구조와 설계에 대해서 자세히 기술한다. 4장은 구현에 관한 부분으로써, 3장에서 제시한 설계 부분을 실제 개발환경에서 구현을 통해 레이아웃 스코어링을 통한 단편화 진단 및 단편화된 파일이 자동적이고 지속적으로 해결되는 모습을 보인다. 5장은 제안된 기법과 기존의 파일 시스템간의 성능을 비교하고, 오버헤드를 측정하였다. 마지막으로 6장에서는 본 연구의 결론과 그 의미를 정리하고, 향후에 수행 되어야 할 연구에 대해서 기술한다.

2. 관련 연구

2.1 FFS(Fast File System) 디스크 할당 정책

FFS은 버클리 대학에서 만든 유닉스 파일 시스템의 한 종류로써 디스크의 대형화에 따라 기존 유닉스 파일 시스템의 문제점을 해결하기 위해 개발되었다. 기존 유닉스 파일 시스템은 메타 데이터와 데이터 블록들의 저장 위치가 물리적으로 떨어져 있었다. 그리고 파일들간의 연관성을 전혀 고려하지 않은 블록 할당 정책을 사용하였다. 이로 인해 헤드의 움직임이 빈번해 지고, 쓰레싱(thrashing)이 일어나게 되었다. 이와 같은 문제점을 해결하기 위한 파일 시스템이 FFS이다. FFS는 기존 유닉스 파일 시스템의 문제점을 해결하기 위해 디스크의 지역성(locality)을 고려하여 같은 실린더 그룹에 메타 데이터와 데이터 블록들을 함께 위치시키고 한 디렉토리 내의 파일에 대해서도 같은 실린더 그룹에 할당하는 정책을 사용함으로써 헤드의 움직임을 줄였고, 파일 시스템의 성능도 향상되게 되었다.

FFS는 하드디스크 드라이브를 연속적인 실린더(cylinder)

들의 집합인 실린더 그룹(cylinder group)으로 나누어 관리한다. 실린더는 디스크의 트랙과 비슷한 개념으로 디스크가 여러 장 있을 경우 수직으로 같은 위치의 트랙들을 합하여 실린더라고 칭한다. 디스크를 실린더 그룹으로 나누어 관리하는 이유는 디스크의 지역성(locality)을 활용하여 관계가 있는 데이터를 같은 실린더 그룹에 저장하기 위해서이다. 이러한 할당 정책은 한 디렉토리 내의 파일들을 같은 실린더 그룹에 위치하게 한다. 지역적 특성을 고려한 할당을 하기 위해 FFS에서는 새로운 파일에 대해 블록을 할당할 때 우선 관계가 있는 파일들이 저장되어 있는 실린더의 빈 공간을 우선 탐색한다. 같은 실린더 내에 빈 공간이 없으면 같은 실린더 그룹 내에서 할당할 블록을 찾고, 같은 실린더 그룹에서도 할당할 여유 공간이 없다면 인접해 있는 실린더 그룹의 실린더에서 빈 공간을 찾아 블록을 할당한다. 이와 같이 FFS의 디스크 할당 정책은 실린더 그룹이라는 지역적인 특성을 고려한 할당을 실시하여 하드 디스크 드라이버의 헤드 움직임을 최소한으로 하려는 알고리즘을 사용하고 있다.[6]

이러한 지역성을 기반한 블록 할당 정책은 기존 파일 시스템에 비해 높은 읽기/쓰기 성능을 얻을 수 있었다. 하지만 하나의 실린더 그룹의 블록이 모두 소모된 이후에도 계속해서 근접한 블록에 대한 쓰기 요청이 들어오면 논리적으로 근접한 블록들이 단편화된다. 연속되지 않았더라도 일단 빈 공간을 찾아 기록을 하는 것이 우선이기 때문이다. 이런 단편화 현상은 파일 시스템이 계속해서 사용됨에 따라 심화되는 경향을 보인다.

기존 FFS를 개발한 McKusick은 단편화 현상을 해결하기 위해 블록 재할당 단계를 포함한 새로운 블록 할당 알고리즘[3]을 기존 FFS에 적용시켰다. 이는 블록 할당 정책에 따라 일단 할당된 블록들을 디스크에 데이터를 쓰고자 할 때 다시 한 번 클러스터링(clustering)하는 알고리즘이다. 이 알고리즘의 사용으로 단편화 현상이 발생하는 빈도는 줄일 수 있었으나, 이미 발생한 단편화 현상은 단편화된 블록들이 속한 파일이 지워지기 전에는 해소되지 않는 문제가 여전히 남아있다.

현재 널리 쓰이는 EXT2와 EXT3 파일 시스템[14]은 위와 같은 FFS를 모태로 하고 있기 때문에 동일한 블록 할당 정책을 가진다. 그리고 파일에 대한 다양한 속성, 보다 긴 파일 이름, 크래쉬 상황에서의 효율적인 복구를 위한 저널링(journalling) 기능 등 현대적인 요구사항을 충족시키는 방향으로 발전되었다. 그리고 단편화 현상에 관해서는 단편화 발생을 줄이고 작은 파일에 대한 보다 높은 읽기/쓰기 속도를 얻기 위해 한 번에 다수의 블록을 할당하는 정책을 사용하고 있다. 이러한 정책의 사용으로 작은 파일에서의 단편화 현상은 크게 줄일 수 있지만, 여전히 이미 발생한 단편화 현상에 대한 해결책은 제안되지 않은 상태이다.

2.2 레이아웃 스코어링 기법

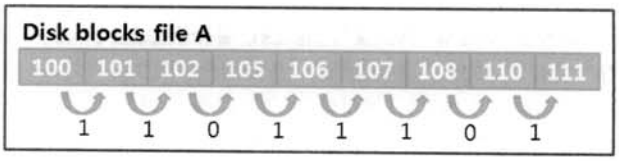
레이아웃 스코어링은 현재 사용 중인 파일 시스템의 단편화 정도를 점수로 나타내는 방법이다. [2]에서 제시된 방식으로, 파일의 단편화 정도를 0부터 1사이의 점수로 나타내는 방식이다. (그림 1)에서와 같이 파일의 블록 할당 연속성을 점수로 측정하는 방식으로 하나의 블록이 앞의 블록과 연속하면(같은 파일에 속한 블록이면) 1, 그렇지 않으면 0으로 점수를 할당한 다음 모든 점수를 더하고 (블록 개수 -1)로 나눈다. 이러한 방식을 사용하여 파일 시스템의 현재 단편화 정도를 측정할 수 있다. 이와 관련하여 Keith A. Smith는 각종 파일 시스템의 단편화 정도를 판단하고 실제 파일 시스템에서 변화의 양상을 파악하는 연구[5] 등에 사용하였다. 그리고 [3]에서는 두 가지 FFS의 블록 할당 알고리즘의 차이점을 파악하는 도구로도 사용하였다. 이러한 연구는 파일 시스템의 단편화 정도를 단순하게 측정할 수 있는 기준은 마련하였지만, 다음과 같은 한계점이 있음을 지적할 수 있다.

첫째, 사용자가 어떤 정책을 가지고 수동으로 단편화 정도를 측정하여야 하며, 사용자가 수행하지 않는 이상 시스템 스스로는 어떤 동작도 취하지 않는다. 대용량 서버의 관리자를 생각해보면, 관리의 초점을 서버가 정상적으로 서비스를 제공하는데 두고, 파일 시스템의 단편화로 인한 성능 저하까지는 일반적으로 고려하지 않는다. 따라서 자동으로 이러한 정보를 측정하고 알려주는 시스템이 요구된다.

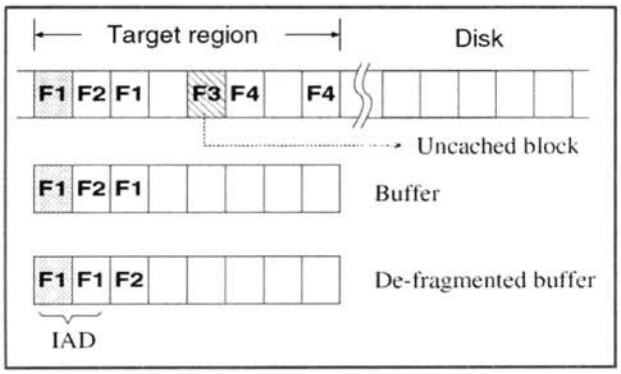
둘째, 단편화 측정으로 인한 오버헤드가 존재하고, 그로 인한 성능 저하는 현재 측정된 바가 없다. 실제 사용되는 메커니즘은 FFS 파일 시스템의 슈퍼 블록과 각 실린더 그룹의 디스크립터 정보를 읽어서 파일 시스템의 블록이 어떻게 할당되었는지 스냅샷(snapshot)을 만든 후, 이 내용을 분석하는 것이다. 이 과정에서 실린더 그룹의 블록까지 모두 접근을 하게 되는데, 이것은 사실상 파일 시스템의 모든 메타 데이터를 필요로 하는 작업이고, 간접 블록의 접근으로 인해 디스크 헤드가 전체 디스크 영역을 모두 움직여야 한다. 만약 버퍼의 도움을 받을 수 없다면, 이 동작은 순간적으로 커다란 I/O의 집중을 야기하게 되며 시스템의 성능 저하를 가지고 올 수 있다. 하지만, 실제로 모든 파일의 간접 블록이 버퍼에 올라와 있을 수는 없을 것이다.

2.3 De-fragmentation File System

[8]에서는 부분적인 단편화 해결 방안을 포함한 파일 시스템을 제시하였다. 이 시스템은 버퍼 캐쉬 내의 블록을 대상으로 재배치를 수행하는 파일 시스템으로, 아이노드(i-node) 단위의 부분적인 단편화 해결 방안을 제시하고 있다. 만약 단편화 현상이 발생한 파일이 버퍼 캐쉬 내에 있다면, 버퍼에서 해당 파일에 대해 조각을 모으고, 배치하여 디스크에 기록하는 방식이다. 이 방식은 추가적인 I/O가 필



(그림 1) 레이아웃 스코어링 방식



(그림 2) 목표구역 안에서의 캐쉬 되지 않은 블록

요치 않다는 장점을 가지지만, 버퍼 캐쉬에 파일의 내용이 로딩이 되지 않거나, 부분적으로 읽기를 위해 사용되는 파일은 결코 단편화 해소가 이루어지지 않는 단점이 있다. 또한, (그림 2)에서 보는바와 같이 단편화 해결을 위해 지정된 목표 구역(target region) 안에서도 버퍼 캐쉬에 올라와 있지 않은 잘 사용되지 않는 파일에 속한 블록이 다른 파일의 중간블록에 할당되어 있다면 단편화 해소를 위한 할당 시 제약을 받게 되고, 단편화 해소는 불완전하게 이루어지게 된다. 추가적인 I/O가 없이 단편화 현상을 많이 해소할 수 있는 기법이기 때문에 큰 장점을 가지고 있지만, 소극적인 정책을 사용하고, 현재 상태를 판단할 수 있는 척도를 가지고 있지 않다. 소극적인 정책을 사용하고 있다고 할 수 있는 근거는 [9, 10, 11]를 통해 제시한다. 특히 [9]는 파일의 실제 입·출력에 대한 정보를 수집하여 실제 디스크에서 파일의 읽고/쓰기가 어느 정도의 비율로 파일이 사용되는가의 수치를 살펴본 연구이다. 이 연구에 따르면 실제 디스크에서는 읽기가 80%, 쓰기가 20%정도 입·출력되고 있으며, 이 수치에서도 [8]에서 제안한 파일 시스템을 적용하여 변화가 일어난 상태의 파일에 대한 쓰기를 진행할 경우, 전체적인 쓰기 20%에서도 80%는 새로운 파일이 쓰기가 되는 것이고, 20% 정도만이 수정이 가해지는 대상파일로 볼 수 있다. 따라서 소극적인 정책을 사용하고 있으며, 이 메커니즘을 이용한 단편화 해결 방식을 활용하기 위해서는 사용자가 어떤 정책을 가지고 수행을 시켜야 한다.

3. 설 계

3.1 ALS(Autonomic Layout-scoring System) 설계

3.1.1 블록 할당 시 스코어링 기법

본 논문에서는 [2]의 레이아웃 스코어링 기법과 동일한 스코어 측정 방법을 사용한다. 특정 아이노드에 대해 디스크 블록을 할당할 때, 앞서 할당된 블록과 연속된 블록이 할당되면 1, 그렇지 않다면 0의 점수를 부여한다. 이러한 스코어링 동작을 파일이 생성되고 데이터가 쓰이는 동안 자동적이면서 지속적으로 측정하기 위해 리눅스 파일 시스템에서 파일이 자라나는 모델을 분석하였다. 리눅스 파일 시스템에서는 크게 두 가지 방식으로 파일에 데이터가 쓰인다.

첫째, 파일이 생성되고 연속적으로 데이터가 쓰이는 일반적인 경우이다. open() 시스템 콜을 O_CREAT 옵션과 함께 호출하면 새로운 아이노드가 할당되고, write() 시스템 콜을 통해 특정 지점부터 데이터를 쓴다.

둘째, 홀(hole)을 이용한 쓰기 기법이다. 파일을 새로 생성한 후, lseek() 함수 등을 이용해 파일의 뒷부분부터 데이터를 쓰는 방식이다. 윈도우즈 운영체제 계열에서는 스파즈(sparse) 파일이라는 방식으로 동일한 기법을 제공한다. 이 두 가지 경우 이외에는 쓰기 수행에 대해 새로이 블록이 할당되지 않기 때문에 고려할 필요가 없다. 만약 기존의 데이터를 수정하면 블록의 내용만이 수정되기 때문이다. 리눅스 파일 시스템의 모델은 VFS 인터페이스에 따라 모든 파일 시스템이 동일하게 지켜져야 하므로, 위와 같은 모델은 모든 파일 시스템에 적용 가능하다.

본 논문에서는 위의 두 가지 모델 가운데 첫 번째 모델을 주된 대상으로 하여 스코어링 기법을 설계하였다. 우선 아이노드에 새로운 필드 두 개를 추가한다. 한 필드에는 앞서 할당된 블록을 기록하고, 다른 한 필드에는 레이아웃 스코어를 기록한다. 그리고 블록이 할당될 때마다 앞서 할당된 블록과 비교하여 연속적이면 1, 그렇지 않으면 0을 레이아웃 스코어에 계속해서 더해 나간다. 이렇게 구한 레이아웃 스코어를 현재 아이노드가 사용 중인 블록의 수로 나누면 현재 아이노드에서 연속적인 블록의 비율을 퍼센티지(percentage)로 구할 수 있다. 블록은 항상 연속적으로 할당되기 때문에 아이노드에 이전에 할당된 블록 번호를 기록하면 파일의 특정 부분에 대한 블록 번호를 다시 구할 필요가 없다. 이는 레이아웃 스코어링을 위해 메타 데이터를 다시 디스크에서 검색해야할 필요를 없애주므로 성능에는 전혀 문제가 없다.

각 블록 그룹의 레이아웃 스코어는 그룹에 속한 아이노드의 레이아웃 스코어를 모두 합한 값이다. 블록 그룹 디스크립터에도 레이아웃 스코어를 위한 필드를 두어 각 아이노드의 스코어를 합한다. 그리고 할당된 모든 블록을 기록하여 스코어 값과 비교하면 아이노드에서와 마찬가지로 연속적인 블록의 비율을 구할 수 있다. 이 동작은 각 아이노드에 대한 새로운 블록 할당이 이루어질 때 동시에 수행된다. 블록 그룹에 대한 가장 직관적인 방식은 그룹에 포함된 블록들이 연속적으로 할당 되었는지 검사하는 것이다. 그러나 이 방식은 블록이 할당되는 방식과 상이하다. 실제 블록이 할당될 때에는 단순히 블록이 할당 되었는지 아닌지에 대한 비트를 검사할 뿐, 해당 블록이 어떤 아이노드에 할당 되었는지 알 수 없다. 이 정보를 얻기 위해서는 해당 블록을 소유한 아이노드를 검색해야 하는데 이러한 동작은 성능에 커다란 저하를 가져오게 된다. 본 논문에서 제안하는 방식을 사용하면 전체 블록에 대해 같은 레이아웃 스코어 결과를 얻을 수 있다. 각 그룹에 대해 정확한 스코어를 얻을 수는 없지만 해당 그룹에 포함된 아이노드들의 스코어를 알 수 있기 때문에, 블록 그룹별로 단편화 해소를 위한 기법을 적용할 수 있다.

전체 디스크에 대한 레이아웃 스코어는 블록 그룹에서의 기법과 유사하게 블록의 할당이 일어나는 경우 그 스코어를 슈퍼 블록 디스크립터에 계속해서 합한다. 이 스코어를 디스크에 할당된 전체 블록의 수로 나누면 전체 디스크에서 연속적인 블록의 비율을 계산할 수 있다. 블록 그룹에서의 방식과 마찬가지로 성능에는 문제가 없다.

3.1.2 블록 해제 시 스코어링 기법

블록이 할당되는 경우 계속해서 스코어를 합하는 것으로 레이아웃 스코어를 구할 수 있다. 이 스코어가 감소하는 경우는 블록이 해제되는 경우인데, 가장 어려운 점은 블록이 해제될 때 이 블록이 연속된 블록이었는지 아닌지를 판단하는 문제이다. 이 문제를 해결하기 위해 본 논문에서는 리눅스 파일 시스템의 파일 모델에서 블록이 해제되는 경우를 분석해 보았다.

첫째로 파일이 삭제되는 경우이다. 파일은 sys_unlink() 시스템 콜에 의해 삭제되는데 ext2_truncate() 함수를 호출하여 아이노드의 모든 할당된 블록을 해제하게 된다.

둘째는 블록이 부분적으로 해제되는 경우이다. 리눅스 파일 시스템의 모델은 파일에 대한 부분적인 삭제를 수행하는 인터페이스를 제공하지 않는다. 그러나 내부적으로는 부분적인 해제가 이루어지는데, SMP(symmetric multiprocessing) 시스템을 고려한 동작에서 할당되었던 블록이 취소되고 다시 할당을 수행할 경우이다.

본 논문에서는 단순히 파일이 삭제될 때 해당 아이노드에 기록 되어있던 레이아웃 스코어와 할당 되어 있던 블록의 개수를 블록 그룹의 정보에서 감소시키는 방식을 사용하였다. 이것은 첫 번째 블록 해제 모델에 부합되는 방식인데, 두 번째 모델에 대해서는 블록 할당 시 스코어를 합하지 않는 방식을 사용하여 해결할 수 있다. 만약 블록을 해제할 때 이러한 것을 고려하려고 하면 임시로 할당, 해제되는 블록들에 대해서도 모두 앞서 할당, 해제된 블록에 대한 정보가 기록되어 있어야 하고 이를 검색해야 하기 때문에 성능의 저하가 발생한다. 따라서 우리는 임시적인 할당 시에 스코어링을 수행하지 않는 방식을 사용하여 문제를 원천적으로 해결하였다.

3.1.3 사용자 인터페이스의 추가

새로운 기능의 추가로 인해 사용자가 얻을 수 있는 정보가 늘어났다. 따라서 우리는 사용자가 ioctl() 인터페이스를 통해 아이노드와 블록 그룹, 그리고 슈퍼 블록의 단편화 관련 정보를 얻을 수 있는 서비스를 추가하였다. ioctl() 시스템콜을 호출할 때, 아래와 같이 정의된 파라미터를 요청 필드에 넣어 주어서 정보를 요청할 수 있다.

"EXT2N_IOC_GET_AGING_COUNT_INODE"는 한 아이노드에 대한 정보를 얻을 수 있다. 이 파라미터를 이용해 ioctl()을 호출하였을 경우, (그림 3)과 같이 출력된다. Blocks는 파일이 사용하고 있는 전체 블록을 나타낸다.

"EXT2N_IOC_GET_AGING_COUNT_SUPER"는 전체 블록 그룹들과 슈퍼 블록에 대한 정보를 출력한다. 예를 들어, 노화가 진행된 파일 시스템에 대한 정보는 (그림 4)와 같다. SCORE는 연속적인 블록을 나타내는 레이아웃 스코어, ALL은 전체 할당된 블록, FREE는 할당되지 않은 블록의 개수를 나타낸다.

```
[root@localhost ~/test]# dmesg
Inode =          14 Continuity Rate = 99%

<Aging count =       10229 Blocks =       10232>
```

(그림 3) 아이노드에 대한 정보

```
<Continuity of Blocks Groups>
No 0 Continuity =  99%  <SCORE = 16150  ALL = 16208 FREE = 13528>
No 1 Continuity =  83%  <SCORE =  4986  ALL =  5942 FREE = 26259>
No 2 Continuity =  85%  <SCORE =  1962  ALL =  2294 FREE = 29997>
No 3 Continuity =  85%  <SCORE =  5029  ALL =  5880 FREE = 26321>
No 4 Continuity =  90%  <SCORE =  5459  ALL =  6049 FREE = 26242>
No 5 Continuity =  74%  <SCORE =  1243  ALL =  1671 FREE = 30530>
No 6 Continuity =  80%  <SCORE =  2289  ALL =  2838 FREE = 29453>
No 7 Continuity =  67%  <SCORE =  1134  ALL =  1679 FREE = 30522>
No 8 Continuity =  82%  <SCORE =  2081  ALL =  2511 FREE = 29780>
No 9 Continuity =  81%  <SCORE = 31548  ALL = 38603 FREE =  5712>
No 10 Continuity =  84%  <SCORE =  3703  ALL =  4397 FREE = 16661>
No 11 Continuity =  84%  <SCORE =  3493  ALL =  4115 FREE =  804>

*** Summary Continuity:  85% Contiguous blocks = 79077 All blocks = 92187
```

(그림 4) 블록 그룹과 슈퍼 블록에 대한 정보

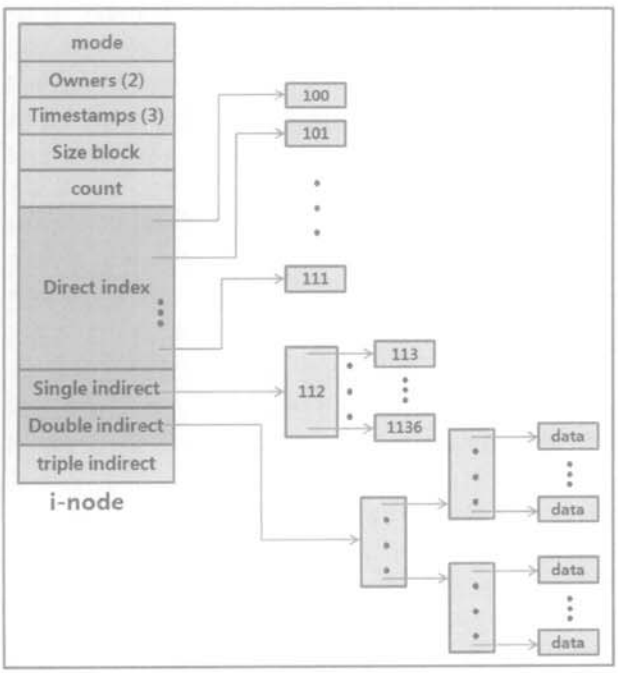
3.1.4 간접 접근 블록에 대한 고려사항

EXT2 파일 시스템은 3단계의 간접 접근 블록(indirect block) 방식을 사용한다. 간접 접근 블록이란, 아이노드에 직접 주소를 기재하지 않고 데이터 블록을 다른 데이터 블록의 주소를 저장하는데 사용하여 보다 많은 데이터 블록을 표현하기 위해 사용되는 기법이다. 간접 접근 블록의 활용은 블록 할당 정책에서 문제점을 발생시킨다. 슈퍼블록이나 아이노드 정보가 포함되어있는 그룹 디스크립터 블록 등의 메타 데이터들은 파일 시스템 생성 시에 해당 파티션에서 특정한 위치에 기록되게 된다. 하지만 간접 접근 블록은 메타 데이터 임에도 데이터 블록 구역에 할당이 되어야 하기 때문에 동적인 블록 할당 정책에 영향을 받게 된다. 만약 어떤 데이터에 접근하고자 할 때, 해당 데이터를 저장하고 있는 데이터 블록과 그 데이터 블록을 가리키는 간접 접근 블록들이 서로 다른 실린더에 있다면 해당 데이터에 접근하는 과정은 헤드의 이동을 필요로 하게 된다. 이는 성능 상의 문제가 되기 때문에 블록 할당 정책은 간접 접근 블록에 대한 고려도 반드시 포함하고 있어야 한다. 만약 데이터 블록만을 이용해 파일의 분산 정도를 살피게 된다면, 순차적으로 완벽한 할당이 이루어진 경우라도 물리적으로 근접해 있지 않은 간접 블록으로의 접근이 필요하기 때문에 실제로 이 파일에 접근할 때 이상적인 성능은 얻을 수 없다. 본 논문에서는 해당 파일에 대한 읽기가 순차적으로 이루어졌을 경우의 성능이다. 따라서 우리는 간접 접근 블록 또한 순차적으로 할당되어 있을 때 비로소 이상적인 블록 할당이 이루어졌다고 정의한다. (그림 5)에서와 같이 12개의 직접 접근 블록이 블록 번호 100부터 111인 블록에 할당되고, 그 이후에 1단계 간접 접근블록을 사용해야 하는 경우, 112번 블록

이 할당되고 그 이후의 데이터 블록은 113번부터 할당이 되어야 이상적이다. 물론 이 112번 블록은 많은 수의 데이터 블록에 대한 주소를 저장하고 있다. 4Kbytes의 블록 크기를 갖도록 생성된 파일 시스템의 경우, 1024개의 데이터 블록 주소를 저장할 수 있다. 만약 할당이 순차적으로 이루어졌다면 112번 블록은 113번부터 1136번 블록에 대한 주소를 저장하고 있을 것이다. 그렇다면 112번과 113번은 연속적이지만 1136번 블록에 대해 접근하려고 할 때는 112번과 1136번은 연속적이지 않다. 그러나 우리는 순차적인 읽기가 이루어지는 것을 가정하였기 때문에 113번 데이터 블록에 접근이 되는 순간, 112번 블록의 데이터는 메모리에 캐싱되어 있음을 알 수 있다. 따라서 1136번 블록에의 순차적인 접근은 112번 블록에 대한 디스크로의 접근 없이 이루어져 성능에 거의 영향을 주지 않는다. 이러한 접근 방식은 보다 다단계의 간접 블록을 사용할 때의 성능을 예측할 때도 동일하게 적용할 수 있다.

3.2 LCD-FS(Lazy-Copy Defragmentation File System) 설계

본 논문에서 제안하는 LCD-FS는 다음과 같은 순서로 동작한다. 파일 시스템에서 파일이 close( ) 되는 순간 해당 파일의 크기와 단편화 여부를 측정하고, 판단하여 단편화를 해결해야할 대상 파일을 선정하고, 선정된 파일의 아이노드를 LCD 데몬(daemon)의 요청 큐에 넣는다. 그리고 단편화된 파일을 새로운 공간에 옮길 수 있도록 연속적으로 빈 공간을 검색하고 예약 할당을 해 둔다. 할당이 끝나면 실제로 데이터를 이동시키는 작업을 수행한다. 이를 위해 I/O 스케줄러(scheduler)에 I/O를 요청하고 대기한다. I/O 스케줄러는 다른 프로세스의 CPU 및 I/O 작업에 최대한 방해가 되지 않도록 LCD 데몬의 처리 우선순위를 최대한 낮게 설정한다. 하지만 이는 운영체제마다 스케줄링 정책이 다를 수 있기 때문에 우리는 I/O 장치의 유휴 시간을 찾기 위해 하드디스크 헤드의 움직임을 검사하는 기법을 제안한다.



(그림 5) 아이노드 내부 구조

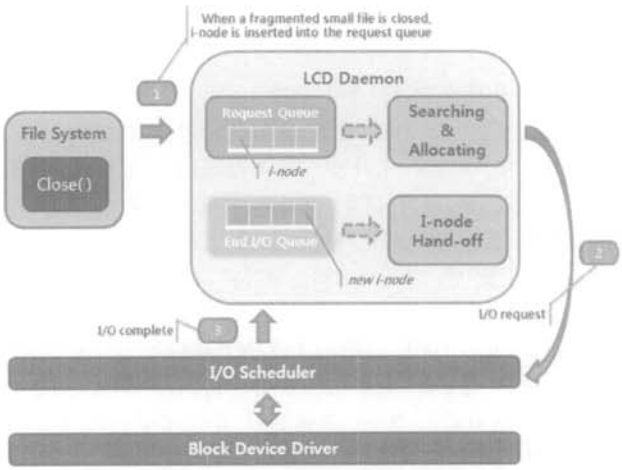
단편화 된 파일의 데이터가 연속된 빈 공간으로 복사된 후에는 이 정보를 아이노드에 반영한다. 이러한 과정을 거쳐 우리는 성공적으로 단편화를 해소시킬 수 있다. (그림 6)에서 이러한 전체 시스템 설계를 표현 하였고 아래에서 각각의 과정을 보다 상세히 설명한다.

3.2.1 대상파일 선정

아이노드 내부구조에서 파일이 직접 접근 블록(direct block) 부분의 할당을 넘어서 간접 접근 블록 영역까지 할당되는 크기의 파일은 13번째 블록에서 순간적으로 읽기/쓰기의 처리량이 감소한다.[2] 이것은 파일의 크기가 13번째 블록 이상의 공간을 차지하는 파일을 대상으로 단편화를 해결하는 것이 오히려 전체적인 시스템의 성능에 해가 되는 것을 의미한다. 이러한 이유로 본 논문에서는 파일의 크기가 48KB(12 블록)이하인 작은 파일을 대상으로 단편화를 해소하고자 한다. 또한, ALS의 정보를 활용하여 단편화가 진행된 상태의 작은 크기의 파일을 선정한다. 이렇게 선정된 단편화 된 작은 파일의 아이노드가 LCD 데몬의 요청 큐에 순차적으로 들어가고 단편화 해소 동작의 수행을 기다린다.

3.2.2 블록의 검색과 할당

단편화된 대상파일의 아이노드가 요청 큐에 들어오게 되면 우선 파일의 크기를 측정하여 대상 파일이 새롭게 할당될 연속된 빈 공간을 찾게 된다. 빈 공간을 찾기 위한 과정은 다음과 같다. EXT2 파일 시스템은 효율적인 디스크 관리를 위해 다수의 블록 그룹을 가지고 있다. 각 블록 그룹에는 블록의 사용여부를 구분하기 위한 블록 그룹별로 비트맵 정보를 특정 블록에 가지고 있다. 이 한 블록에는 32768개(4KB = 4096Byte =32768bit)의 블록에 대한 정보가 포함되어 있으며 이러한 비트맵 정보의 분석을 통해 단편화된 파일크기 만큼의 새로운 연속된 빈 공간을 찾을 수 있다. (그



(그림 6) LCD-FS 설계

(그림 7) 블록 할당 정책

림 8)의 ii)는 비트맵의 정보를 통해 단편화된 대상 파일 크기만큼의 연속된 빈 공간을 찾고 할당하는 과정을 보여준다.

연속된 빈 공간을 찾을 때에도 지역성(locality)을 고려하여 대상파일이 위치했던 곳에서 가장 가까운 물리적 공간에서부터 찾기를 시작하는 방법을 사용한다. (그림 7)에서 보는 바와 같이 우선 메타 데이터와 데이터 블록들의 사이의 공간부터 검색을 하고 해당 위치에 빈 공간이 없을 시에는 메타 데이터의 앞쪽 공간을 찾고, 다음에는 데이터 블록의 뒤쪽 공간을 찾는 식의 방법을 이용하여 기존의 EXT2 파일 시스템의 디스크 할당 정책을 최대한 유지하도록 하였다. 지역성을 고려하지 않고 빈 공간을 찾을 경우 블록 그룹의 지역적인 할당 정책에 방해요소가 될 수 있고, 오히려 디스크 헤드의 움직임이 더 빈번하게 늘어나 전체적인 성능 저하를 야기 시킬 수 있다.

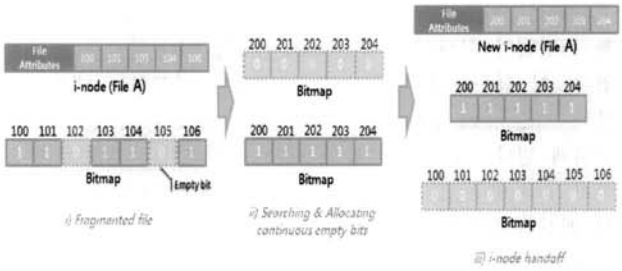
3.2.3 지연 복사의 수행

위와 같은 과정을 통해 연속된 빈공간이 확보 되었다면 다음 과정에서는 실제 복사가 이루어져야 한다.. 이를 위해 I/O 스케줄러에게 I/O 요청을 하게 되는데 여기에서 고려되어야 하는 것은 어느 시기에 실제 복사가 일어나는 것이 다른 프로세스의 CPU 및 I/O 작업에 최대한 방해가 되지 않느냐는 것이다. 이 문제점을 해결하기 위해 본 시스템에서는 우선 LCD 데몬 처리의 우선순위를 가장 낮게 할당하여 처리하게 한다. 그리고 디스크의 헤드가 일정시간 동안 동작하지 않으면 그 시점을 유휴 시간으로 인식하여 해당 시점에 하나의 아이노드에 대한 I/O 요청을 한다. 우리는 12블록(최대 48KB) 이하의 작은 파일에 대해 단편화를 수행하므로 해당 I/O 요청은 I/O 컨트롤러가 처리하는 가장 작은 양인 64KB 보다 적다. 따라서 해당 I/O 요청은 가장 최소한의 오버헤드를 발생시킨다고 간주할 수 있다. 이를 통해 다른 프로세스의 I/O 작업을 최대한 방해하지 않을 수 있다.

이렇게 I/O의 유휴 시간을 기다려 최소한의 I/O 동작을 수행하는 기법을 우리는 지연 복사 기법이라고 명명하였다.

3.2.4 아이노드의 수정

요청된 I/O 작업이 모두 완료되면 I/O 스케줄러는 이를 LCD 데몬에 알려준다. 현재는 데이터가 기존에 할당된 블록들과 새롭게 할당된 연속적인 블록들에 이중으로 존재한다. 따라서 아이노드 정보를 수정하여 아이노드가 새롭게 할당된 블록을 가리키도록 수정하여야 한다. 이를 위해 기존에 할당된 블록들을 각각의 블록 그룹에 맞춰 프리 블록으로 해제하며 새롭게 할당된 블록을 아이노드 정보에 기록한다. 이 과정에서 다른 프로세스가 아이노드에 접근하지 않도록 보호하여야 한다. 이러한 과정을 우리는 아이노드



(그림 8) LCD Daemon의 동작모습

핸드오프(hand-off)라고 부른다. 이 같은 내용은 (그림 8)의 iii)에 잘 나타나 있다.

4. 구 현

4.1 개발 환경

우리는 3장에서 설계한 ALS와 LCD-FS를 리눅스 2.6.18 버전의 EXT2 파일 시스템에 구현하였다. EXT2 파일 시스템은 UNIX의 FFS(Fast File System)을 기반으로 하여 개발된 파일 시스템으로 현대 파일 시스템의 기초가 되는 가장 널리 쓰이는 파일 시스템이다. 현재 EXT3 파일 시스템까지 개발되어 있는데, EXT3는 EXT2를 기초로 저널링(journalling) 기능을 추가한 것이다. 저널링 기능은 파일 시스템의 하부에 저널링 계층을 두어 추가하였기 때문에 실제 동작과 구조는 EXT2 파일 시스템과 크게 다르지 않다. 리눅스에서 파일 시스템은 동적으로 로드 가능한 커널 모듈(LKM: Loadable Kernel Module)로 구현할 수 있고, 시스템의 재시작 없이도 파일 시스템의 교체가 가능하다. 따라서 우리는 ext2 파일 시스템을 모듈화하여 새로운 기능을 추가하였다. 커널의 다른 부분에 대해서는 수정을 가하지 않았기 때문에 새로운 기능을 활용하는데 커널 컴파일이나 재시작이 필요치 않다. 이는 새로운 파일 시스템을 배포하고 실제로 활용하는데 매우 중요한 요소이다.

4.2 ALS 기법

4.2.1 블록 할당 시 스코어링 기법

블록 할당 시의 스코어링 기법을 구현할 때 가장 큰 어려움은 실제 블록 할당의 과정에서 SMP의 고려로 인해 코드가 복잡하고, 임시로 할당되는 부분에 대한 고려가 있어야 한다는 점이다. 이러한 코드의 동작에 맞추어 임시로 스코어를 측정한 후 실제로 블록의 할당이 아이노드에 확정되는 순간 레이아웃 스코어를 합하도록 구현하였다. 여기에서 언급되지 않은 문제점은 직전에 할당된 블록 정보를 어떻게 처리할 것인가 하는 것이다. 이 문제점 또한 임시 변수를 사용하여 해결하였으며, 실제 정보는 ext2_inode_info 구조체에 추가하였다. 그러나 이 구조체는 메모리에만 존재하고, 실제로 디스크에는 기록되지 않는다. 따라서 만약 해당 파일에 대한 아이노드 정보가 메모리에서 해제 되었다가 다시

로드되면, 이전 할당 블록의 정보는 사라지게 된다. 이 문제를 해결하기 위해서 우리는 ext2_find_goal() 함수를 수정하였다. 이 함수는 다음에 할당할 블록을 설정하는 내용으로 본래 ext2_inode_info 구조체의 i_next_alloc_goal 필드를 참조하게 된다. 하지만 이 또한 메모리 내에만 존재하는 필드이기 때문에 메모리에서 지워지면 다음에 할당할 블록을 다시 계산하여야 한다. 이를 위해 ext2_find_near() 함수를 호출하게 되고, 이 함수는 목표가 되는 블록의 바로 직전 블록이나 직전의 간접접근 블록을 검색한다. 따라서 이 값을 이용하면 이전 할당 블록의 번호를 구할 수 있다. 이러한 수행을 통해 앞서 언급하였던 ext2_inode 구조체의 필드 부족 문제를 해결하였다.

4.2.2 블록 해제 시 스코어링 기법

블록 해제 시의 스코어링 기법은 할당 할 때보다 쉽게 이루어진다. 앞서 설계 시 언급 하였듯이, 전체 아이노드 블록의 할당이 해제될 경우만 고려하면 되기 때문에 ext2_truncate() 함수만을 수정하였다. 수정한 내용은 함수의 초반부에서 그룹 블록과 슈퍼블록에 대한 디스크립터를 구하고, 할당이 해제된 아이노드의 레이아웃 스코어와 전체 블록 개수를 각각 감소 시켜주는 내용이다. 최대한 단순화한 설계로 인해 간단한 구현으로 문제를 해결할 수 있었다.

4.2.3 관련 구조체의 수정

구조체 내용의 수정은 기존 시스템에 영향을 주지 않아야 하기 때문에 커널의 수정에서는 중요한 내용이다. 그리고 EXT2와 관련된 헤더파일 중 ext2_fs.h는 /include 폴더에 포함되어 있기 때문에 모듈화 시켰을 때 함께 컴파일 되지 않는다. 따라서 이에 포함된 내용들도 수정해서는 안 된다. 여기에는 디스크 i-node, 블록그룹, 슈퍼블록의 레이아웃을 정해놓은 구조체인 ext2_inode, ext2_group_desc, ext2_super_block 등이 있다.

본 논문에서는 i-node의 메모리 구조를 나타낸 ext2_inode_info 구조체에 아래와 같은 필드를 추가하였다.

```
__u32    i_aging_count;              //전체 레이아웃 스코어
__u32    i_aging_count_temp;         //임시 레이아웃 스코어
__u32    i_previous_alloc_block;     //이전에 할당된 블록
__u32    i_previous_alloc_block_temp; //임시로 사용하는
                                      이전 할당 블록
```

위에서 언급한 ext2_fs.h에 포함된 구조체에 대해서는 예약된 블록만을 사용하였기 때문에 추가적인 수정 없이 사용할 수 있다. 이를 통해 새로운 파일 시스템이 모듈로 구성되어 기존 시스템에는 아무런 변화 없이 추가될 수 있다.

4.3 LCD-FS 기법

우리는 3장에서 설계한 LCD-FS를 리눅스 버전 2.6.18의 EXT2 파일 시스템에 구현하였다. 설계에서 제시된 바와 같이 기존 커널을 수정하지 않고 구현하고자 하였으며, 그 결과로 우리는 리눅스 커널 모듈(LKM: Linux Kernel Module) 형태로 새로운 파일 시스템을 제작하였다. LKM 형태로 제작된 경우, 동작 중인 리눅스 시스템에 커널의 수정이나 재컴파일 없이 LCD-FS를 바로 적재하여 사용할 수 있다. 이러한 목표로 인해 구현에 제약을 받은 점은 LCD 쓰레드의 스케줄링이다. LCD-FS는 단편화 해소를 위한 동작을 위해 다른 프로세스의 CPU 및 I/O 작업을 최대한 방해하지 않아야 하기 때문에 LCD 쓰레드의 CPU 및 I/O 스케줄링 우선순위를 최대한 낮출 필요가 있다. 그러나 리눅스 버전 2.6의 CPU 스케줄러는 프로세스들이 최대한 공평하게 수행되는 것을 목표로 하기 때문에 이런 목표를 정확히 달성하기가 어렵다. 따라서 현재는 LCD 쓰레드의 nice 값을 19로 설정함으로써 일반 프로세스 중 가장 낮은 우선순위를 가지도록 구현되었다. 그리고 또한 쓰레드 동작 중 큐에 요청이 없는 경우 yield()를 수행하도록 함으로써 CPU의 사용을 최소한으로 제한하였다. 가장 좋은 형태는 CPU의 유휴 상태에서만 실행되는 유휴(idle) 프로세스를 대체하거나 해당 우선순위로 동작시키는 것이지만 LKM을 위해 제공되는 인터페이스로는 그러한 구현을 수행할 수가 없다. 만약 커널을 수정하여 구현한다면 가능하겠지만 이러한 방식으로 구현할 경우 실제 사용하고자 할 때 커널의 수정 및 재 컴파일, 시스템 리부팅 등이 요구되기 때문에 우리는 최대한 LKM으로 구현하고자 하였다.

두 번째 제약은 I/O 스케줄러의 우선순위 동작이 여러 스케줄러에 따라 서로 다르다는 점이다. 리눅스는 완전 공평 큐잉 (CFQ:Complete Fairness Queueing), 예측(Anticipatory), 데드라인(Deadline), 무동작(No-operation) 스케줄러 등 네 가지 스케줄러를 제공하며 디스크 파티션 별로 서로 다른 스케줄러를 지정해서 사용할 수 있다. 그러나 각각의 스케줄러마다 I/O 우선순위를 처리하는 방식이 틀리며, CFQ를 제외한 다른 세 가지 스케줄러들은 I/O를 가장 느리게 처리하는 Idle 우선순위를 무시한다. 따라서 단순히 I/O 우선순위만을 조정하는 방법으로는 다른 프로세스의 I/O 작업을 방해하지 않는 것을 보장할 수 없다.

우리는 이러한 리눅스의 제약을 극복하기 위해 LCD-FS의 설계에서도 이를 고려하여 디스크 헤드의 움직임을 추적하여 유휴 시간을 찾는 구조를 제안하였다. 리눅스에서는 디스크 헤드의 현재 위치를 얻기 위해 각 디스크에 요청된 I/O 작업을 관리하는 자료 구조인 struct request_queue 구조체에서 end_sector 변수를 활용하였다. I/O 작업이 수행될 때마다 이 값이 업데이트되며 우리는 일정 시간 동안 해당 값의 변화가 없으면 I/O가 유휴 상태라고 판단하였다.

LCD-FS에서 I/O를 요청하는 경우는 세 가지가 있다. 첫째, 연속하여 비어있는 블록을 찾기 위해 블록 비트맵을 읽는 작업, 둘째, 단편화 해소를 위해 쓰기를 수행하는 작업, 셋째, 기존 아이노드에 할당되어있던 블록을 해제하거나 단편화 해소가 실패하여 블록 비트맵을 복원하기 위한 읽기 작업들이다. 리눅스의 I/O 스케줄러들은 대체로 쓰기보다 읽기 작업을 더 빠르게 처리하도록 설계되어 있기 때문에 블

록 비트맵을 읽는 작업들이 기존 I/O의 작업에 더 큰 영향을 줄 수 있다. 우리는 세 가지 경우 모두를 수행하기 전에 유휴 상태를 점검하도록 구현하여 최대한 다른 프로세스의 I/O 작업을 방해하지 않도록 하였다.

또 하나의 고려 사항은 쓰레드의 개수이다. 우리는 한 개의 쓰레드를 사용하는 방식으로 구현하였지만, 초기의 구현은 두 개의 쓰레드를 이용하여 I/O 작업의 요청과 이후의 처리를 각각 수행하는 방식을 사용하였다. 그러나 앞서 언급한 CPU 스케줄러의 한계로 인해 두 쓰레드가 수행되는 경우 하나의 쓰레드를 사용할 때보다 더 많은 CPU 시간을 소요하게 된다는 점을 파악하였다. 이 때문에 최종적으로는 하나의 쓰레드를 사용하는 구조로 구현하였다.

5. 실 험

이 장에서는 EXT2 파일 시스템과 본 논문에서 제안한 LCD-FS과의 성능을 비교 평가하였다. 이를 위해 총 3개의 실험을 진행 하였다. 3개의 실험은 각각 1) EXT2 파일 시스템과 LCD-FS의 레이아웃 스코어링 측정 2) 특정 시점에서의 읽기/쓰기 성능 측정 실험, 3) LCD-FS 동작으로 인한 오버헤드 측정 실험이다.

5.1 실험 환경

CPU	: Intel(R) Pentium(R) 4 CPU 2.6GHz
RAM	: 1GB
HDD	: 61.5GB, 7200RPM
Partition	: 1.4Gbytes, Block groups(12), Block size(4096 bytes), Blocks(364,266), Block per group(32,768)

5.2 단편화 현상의 생성

본 논문에서는 EXT2 파일시스템과 LCD-FS과의 성능 비교를 위한 환경조성을 위해 실제 하드디스크 파티션(partition)을 사용하여 단편화 현상을 생성하였다. [2]에서 K.A. Smith는 실제 사용되는 서버의 파일 시스템에 발생한 작업(workload)들의 로그를 기록하고 이를 다시 수행시켜볼 수 있는 실험 도구를 개발하고 제공한다.[16] 이 도구를 사용하면 실제로 파일 시스템에 행해진 파일의 생성과 쓰기, 삭제, 디렉토리 생성 등을 새로운 파티션에서 빠르게 다시 수행시켜볼 수 있다. 이를 통해 파일 시스템의 노화를 실제로 수행해볼 수 있고, 단편화의 양상을 관찰할 수 있다. [2]에서는 NFS(Network File System)으로 연결된 502MBytes 파티션에 대해 수행된 10개월간의 파일 시스템 워크로드를 수집하였다. 그 결과로 80만개의 파일 시스템 오퍼레이션들이 기록되었고, 이 워크로드는 처음에 전체 파티션 중 10% 가량을 사용하지만, 실행 중간에는 최대 90% 까지 사용률이 증가하고 마지막에는 70% 가량을 차지한다.

본 논문에서는 위와 같은 방법을 파일 워크로드(File work loads)라 명명한다. 파일 워크로드의 진행과 파일의 쓰기와 삭제를 병행 사용하여 새로운 파일시스템에 단편화 현상을 인위적으로 발생시켰다. 실험 대상이 되는 파티션은 1.4 GB의 용량을 갖도록 하였는데 이는 단편화 현상을 보다 심화시키기 위해서이다.

5.3 EXT2 파일 시스템과 LCD-FS과의 성능 비교

실험 1에서는 1.4GB의 공간을 가진 파티션에 5.1에서 언급했던 파일 워크로드를 이용하여 디스크를 노화 시키고, 추가적인 쓰기와 지움을 반복하면서 성능을 비교할 환경을 인위적으로 조성하였다. 이렇게 조성된 환경에 EXT2 파일 시스템과 LCD-S을 적용하여 동일한 조건에서 성능 비교 실험을 실시하였다.

실험 2에서는 (그림 9)에서의 File Operation1 지점과 File Operation10 지점에서 EXT2 파일시스템과 LCD-FS과의 읽기/쓰기 성능을 비교하는 실험을 실시하였다. 이 실험은 파일 크기에 따른 읽기/쓰기 성능을 측정하는 것으로써, 16KB의 크기부터 32768KB의 크기까지의 파일을 대상으로 적용한다.

5.3.1 레이아웃 스코어링 비교

(그림 9)는 <표 1>에서 제시한 방법을 사용하여 각 단계별로 디스크에 해당 동작을 실행하면서 EXT2 파일 시스템과 LCD-FS의 레이아웃 스코어링을 기록한 그래프로써 시간이 지남에 따라 크기의 차이는 있지만 작게는 File Operations 1 지점에서 2.4% 크게는 File Operations 10 지점에서 10.4% 정도까지 레이아웃 스코어링의 차이가 나타나는 것을 보이고 있다. 각각의 지점에서 동작이 완료되면 dd(disk to disk copy)를 이용하여 이미지로 만들어 놓음으로써 실험 3을 위한 자료로 사용하였다. 실험결과에서 보이고 있는 레이아웃 스코어링은 전체 디스크에 대한 결과가

〈표 1〉 단계별 File Operation의 동작

File Operations	동 작	Files / Small files	Used Disk
1	파일 워크로드 1회 실행	8757 / 3369	26%
2	파일 워크로드 2회 실행	17514 / 6738	52%
3	파일 워크로드 3회 실행	26271 / 10107	78%
4	파일 1개 복사, 디렉토리 1개 복사	32013 / 12649	94%
5	파일 1개·디렉토리 2개 삭제, 파일 2개 복사, 파일 워크로드 1회 실행	35281 / 14560	69%
6	디렉토리 3회 삭제	18058 / 6934	53%
7	디렉토리 2개 삭제, 디렉토리 5개 복사 파일 워크로드 1회 실행	46491 / 19546	79%
8	디렉토리 3개 복사	57973 / 24630	90%
9	디렉토리 6개 삭제	34737 / 14364	68%
10	디렉토리 1개, 디렉토리 묶음 삭제 파일 워크로드 1회 실행	17514 / 6738	52%

(그림 9) EXT2-FS와 LCD-FS의 레이아웃 스코어링 비교

아닌 작은 파일(12블록 이하)을 대상으로 한 결과이다. 비록 수치상으로는 근소한 차이를 보이고 있지만 기존의 EXT2 파일 시스템에서 블록할당 시 8개의 블록에 대해서는 미리 예약을 해놓는 정책을 사용하는 것을 감안한다면 (그림 9)에서 보이는 LCD-FS의 레이아웃 스코어링 수치가 상당한 효과를 발휘하는 것임을 판단할 수 있다.

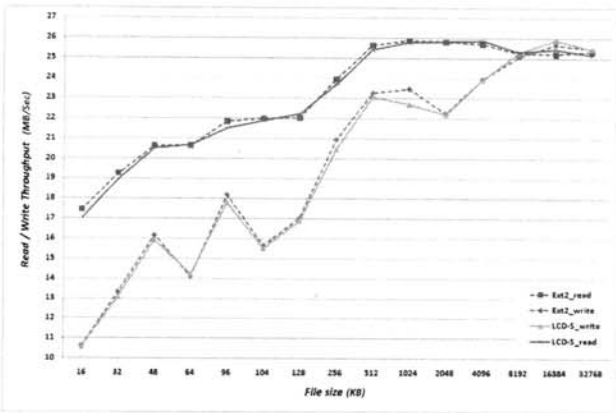
5.3.2 읽기/쓰기 성능 측정

(그림 10)는 (그림 9)에서의 File Operation1의 지점에서 EXT2 파일 시스템과 LCD-FS의 읽기/쓰기 성능을 비교해 놓은 그래프이다. x축은 16KB에서부터 32768KB 크기까지의 파일을 나타내며 해당 파일들을 대상으로 실험 1에서 dd로 만들어 놓은 이미지를 1.4GB의 실험용 디스크에 적용시키고, EXT2 파일 시스템과 LCD-FS을 각각 적용하여 16KB에서부터 32768KB 크기까지의 파일들을 파일 크기 당 10회씩 읽기/쓰기를 반복하여 평균을 낸 수치를 y축에 나타내고 있다. (그림 10)에서 보는바와 같이 파일 워크로드를 1회 실시한 후의 성능은 거의 비슷한 것을 볼 수 있다. 특히 48KB 크기의 파일에서 64KB로 쓰기 성능이 급격하게 떨어지는 부분은 파일이 간접 블록 영역으로 가면서 성능에 영향을 주는 모습을 보이고 있다.

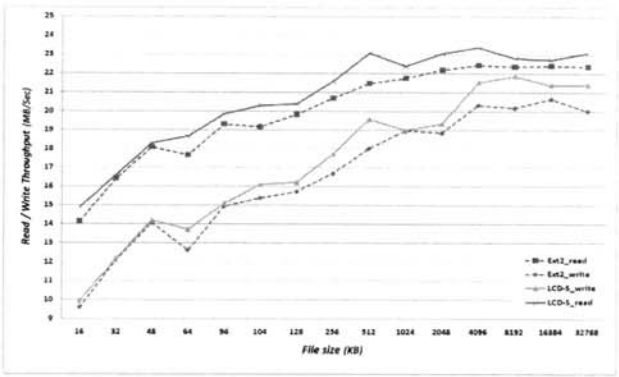
(그림 11)은 (그림 8)에서의 File Operations10의 시점에서 EXT2 파일 시스템과 LCD-FS의 읽기/쓰기 성능을 비교해 놓은 그래프이다. 실험 방법은 File Operations1의 지점에서 이루어졌던 방법과 동일한 방법을 적용하였으며, LCD-FS을 적용하였을 때를 EXT2 파일 시스템의 읽기/쓰기 성능과 비교하여 쓰기성능에서는 1%-8.5%, 읽기에서는 1.2%-7.5% 정도의 향상된 성능을 결과로 보였다. 이러한 결과는 LCD-FS이 파일 시스템의 단편화를 지속적으로 해결하였기 때문에 나올 수 있는 결과로 볼 수 있다.

5.4 LCD-FS의 오버헤드 측정

실험 3에서는 실험 1을 통해 도출하였던 LCD-FS의 성능 향상이 오버헤드를 발생시키는가의 여부를 측정해 보았다. 실험방법은 (그림 9)의 File Operation10 지점에서 EXT2 파일 시스템의 디스크 상태 이미지를 디스크에 적용시키고,



(그림 10) File Operation1에서의 읽기/쓰기 성능 비교



(그림 11) File Operation10에서의 읽기/쓰기 성능 비교

EXT2 파일 시스템과 LCD-FS을 각각 적용하였다. File Operation10 지점에서 EXT2 파일 시스템의 디스크 상태에 추가하여 리눅스 소스 디렉토리를 실험 디스크에 복사하고, grep 명령을 수행하여 검색에 걸리는 시간을 측정하고 비교하였다. 검색을 끝낸 후에는 레이아웃 스코어링도 비교해 보았다. <표 2>에서 보는 바와 같이 리눅스 소스 트리의 grep 명령어 수행 결과에 걸리는 시간은 EXT2 파일 시스템이나 LCD-FS과 커다란 차이를 보이지 않았다. 이는 LCD-FS의 동작이 오버헤드를 발생시키지 않는다는 것을 보여준다. 또한, 레이아웃 스코어링의 측정 결과는 오히려 LCD-FS을 사용하였을 때 단편화가 해소되는 수치를 통해 알 수 있다.

현재 실험은 단편화 현상의 발생을 확인하기 위해 실험에 사용된 디스크를 이용하여 수행되었지만, 여기서 확인할 수 있는 결과는 언제나 동일하다고 할 수 있다. 실제 grep이 수행되는 동안에는 LCD-FS의 단편화 해소 동작은 수행되

〈표 2〉 EXT2 파일 시스템과 LCD-FS의 오버헤드 측정

	EXT2		LCD-FS	
Linux source tree (files=22,068)의 grep 시간 측정	real	23.578s	real	23.614s
	user	0.200s	user	0.184s
	sys	1.064s	sys	1.120s
Layout Scoring 측정	0.990904		1.000000	

지 않기 때문이다. grep이 모두 수행된 이후, 유휴 시간이 발생하면 그때 단편화 해소 작업이 수행된다. 현재 결과를 볼 때, 대략 1% 가량의 단편화가 발생하였으나 LCD-FS에서는 이를 해소해주고 있다. 만약 보다 단편화 현상이 심화되더라도 이를 검출해내는 단계와 해소하는 단계가 명확하게 분리되어 있고 유휴 시간을 활용하기 때문에 오버헤드에는 변화가 없다.

6. 결론 및 향후 과제

본 논문에서는 파일 시스템의 노화로 인해 발생하는 단편화 현상을 해결하기 위해 자동적이고 지속적인 단편화 해소 시스템을 설계하고 구현하였다. 이를 위해 파일 시스템의 단편화 정도를 측정하는 ALS와 실제 디스크에서 발생하는 단편화 된 파일을 새로운 빈 공간으로 복사를 진행하여 단편화를 해소하는 LCD-FS를 제안하였다. ALS를 통해 단편화 정도를 측정하고 단편화가 진행된 파일 중에서 작은 파일(12블록 이하)을 대상으로 아이노드 정보를 LCD 데몬에 넘겨주고 LCD 데몬에서는 단편화 된 파일 크기 만큼의 새로운 연속된 빈 공간을 블록그룹의 비트맵 정보를 이용하여 예약 할당을 해놓고 I/O 스케줄러에게 처리를 요구한다. 이때 I/O 스케줄러는 다른 프로세스의 CPU 및 I/O 작업에 방해가 되지 않게 하기 위해 LCD 데몬의 처리 우선순위를 가장 낮게 설정하고, I/O 장치의 유휴시간을 찾기 위해 하드디스크 헤드의 움직임이 일정시간 동안 없으면 I/O를 처리하게 하였다. I/O 스케줄러에서 LCD 데몬의 처리가 완료되면 아이노드 핸드오프를 통해 아이노드의 정보가 업데이트(update) 되면서 자동적이고 지속적인 단편화 해소가 되는 것이다. 그 결과로 파일의 오퍼레이션들에 대한 레이아웃 스코어가 2-10% 정도로 올라가는 결과를 얻었다. 또한, 읽기/쓰기 성능에서도 기존의 EXT2 파일 시스템보다 LCD-FS을 적용하였을 때 쓰기 성능은 1%~8.5%, 읽기 성능에서는 1.2%-7.5% 정도 향상되는 결과도 보았다.

향후 연구 과제로는 새로운 빈 공간을 찾는 동작에서 과연 단편화 된 파일을 어디로 옮기는 것이 효과적인가 하는 가에 대한 문제이다. 예를 들어 실린더 그룹내에서 단편화 된 파일이 있을 경우 이 파일의 단편화를 해결하기 위해 해당 실린더 그룹이나 인접 실린더 그룹에 여유 공간이 없을 경우 과연 이 파일의 단편화를 해소하기 위해 연관성이 없는 실린더 그룹으로 이동시키는 것이 옳은 것인지 아니면 약간의 단편화를 감수하더라고 지역성을 파괴하지 않고 유지하는 것이 옳은 정책인가에 대한 추가적인 연구가 진행되어야 한다.

참 고 문 헌

[1] Windsor W. Hsu, Alan Jay Smith and Honesty C. Young, "The Automatic Improvement of Locality in Storage Systems," ACM Transactions on Computer Systems, Vol.23, No.4, November, 2005, pp.424-473, 2005.

[2] Keith A. Smith and Margo I. Seltzer, "File System Aging - Increasing the Relevance of File System Benchmarks," In Proceedings of the 1997 ACM SIGMETRICS Conference, pp.203-213, June, 1997.

[3] Keith A. Smith and Margo I. Seltzer "A Comparison of FFS Disk Allocation Policies," 1996 USENIX Annual Technical Conference, pp.15-26, 1996.

[4] Marshall K. McKusick, William N. Joy, Samuel J. Leffler and Robert S. Fabry,"A Fast File System for UNIX," ACM Transactions on Computer Systems, Vol.2, No.3, pp.181-197, 1984.

[5] Keith A. Smith and Margo I. Seltzer, "File Layout and File System Performance," Harvard University Computer Science Department Technical Report.

[6] Marshall K. McKusick, Geoge V. Neville-Neil, "The Design and Implementation of the FreeBSD Operating System," Addison-Welsey, Reading, MA. 2005, pp.362-378.

[7] Abraham Silberschatz, Peter B. Garvin, Greg Gagne, "Operating Systems Principles," John Wiley & Sons, 2006, Appendix A pp.834-842.

[8] W.H. Ahn, et al., "DFS: a de-fragmented file system," Modeling, Analysis and Simulation of Computer and Telecommunications Systems, 2002. MASCOTS2002. Proceedings. 10th IEEE International Symposium on, 2002, pp.71-80.

[9] W. Vogels, "File system usage in Windows NT 4.0," Proceedings of the seventeenth ACM symposium on Operating systems principles, 1999, pp.93-109.

[10] Nitin, A., et al., "A five-year study of file-system metadata," Trans. Storage, 2007, 3(3): p.9.

[11] Sarr Blumson, "Workload Characterization in a Large Distributed File System," CITI Technical Report 94-3.

[12] L. McVoy and S. Kleiman. "Extent-like performance from a UNIX file system," In 13th CM Symposium on Operating Systems Principles, pages pp.137-144, October, 1991.

[13] E. Riedel, C. van Ingen, and J. Gray. "A performance study of sequential I/O on Windows NT 4," In Proceedings of the second USENIX Windows NT Symposium, Seattle, Washington, August, 1998.

[14] Card, R., T. Ts'o, and S. Tweedie, "Design and implementation of the second extended filesystem," 1994.

[15] 이경재, 안우현, 오재원, "디렉토리 지역성을 활용한 작은 파일들의 모아 쓰기 기법", 정보처리학회논문지A, Vol.15-A, No.5, pp.275-286, 2008. 10.

[16] The source code for the aging tool and benchmarks. Available : http://www.eecs.harvard.edu/~keith

[17] Bovet, D.P. and M. Cesati, "Understanding the Linux Kernel," 2003, O'Reilly & Associates.

[18] Love, R., "Linux Kernel Development," 2003: Sams Pub.

이 준 석

e-mail : jslee@os.korea.ac.kr

2004년 육군3사관학교 전산정보처리학과 (학사)

2009년 고려대학교 컴퓨터학과(공학석사)

2009년~현 재 육군3사관학교 교수부 컴퓨터공학과

관심분야 : 운영체제, 파일 시스템

박 현 찬

e-mail : hcpark@os.korea.ac.kr

2004년 고려대학교 컴퓨터학과(학사)

2004년~현 재 고려대학교 컴퓨터학과 석·박사통합과정

관심분야 : 운영체제, 임베디드 소프트웨어, 플래시 파일 시스템

유 혁

e-mail : hxy@os.korea.ac.kr

1982년 서울대학교 전자공학과(학사)

1984년 서울대학교 전자공학과(공학석사)

1986년 University of Michigan 컴퓨터공학과 (공학석사)

1990년 University of Michigan 컴퓨터공학과 (공학박사)

1990년~현 재 고려대학교 컴퓨터공학과 교수

관심분야 : Real-time OS, 임베디드 소프트웨어, 네트워크, 미디어 처리