단편화 해소를 위한 자동적인 레이아웃 스코어링 기법

이준석[O] , 박현찬 , 유혁
고려대학교 컴퓨터학과
{jslee, hcpark, hxy}@os.korea.ac.kr

Autonomic Layout-Scoring technique for File System Defragmentation

Junseok Lee[O] Hyunchan Park Chuck Yoo
Dept. of Computer Science and Engineering, Korea University

요 약

이차적인 저장공간으로 사용되는 하드디스크의 용량은 기술의 발전으로 나날이 커져가고 있다. 이렇게 디스크 용량이 증가함에 따라 파일 시스템의 단편화 현상은 더더욱 심화되며, 전체적인 시스템의 성능을 떨어뜨리는 주원인이 되고 있다. 우리는 단편화 현상 해결을 위한 첫 단계로 자동적이고 지속적인 단편화 측정을 위한 자동적인 레이아웃 스코어링(Autonomic Layout Scoring)기법을 제안한다. ALS 기법을 활용하면 파일 시스템이 생성된 후, 계속해서 단편화 현상의 정도를 측정하게 되며 이를 토대로 단편화 현상을 제거하기 위한 기법을 적용할 수 있다. 본 논문에서는 ALS 기법을 설계하고 리눅스 2.6 버전의 EXT2 파일 시스템에 구현하였다. 그리고 실제 파일 시스템을 수행하면서 단편화 현상을 측정하는 과정과 아이노드, 블록 그룹, 슈퍼 블록이 각각 노화되는 정도를 측정한 결과를 보인다.

1. 서 론

이차적인 저장 공간으로 널리 사용되는 하드디스크의 용량은 기술의 발전으로 나날이 커져가고 있다. 그에 따라 사용자가 하드디스크에 저장해두고 사용하는 파일의 수와 양도 계속해서 증가하고 있다. 하지만, 이러한 파일 사용의 증가는 파일 시스템의 단편화(fragmentation) 현상을 심화시키고, 성능을 떨어뜨리는 주원인이 된다. I/O의 성능은 전체 시스템의 병목현상을 유발하는 주원인이기 때문에 이를 담당하는 파일 시스템의 성능 저하는 시스템의 전체 성능에 큰 영향을 끼친다. 단편화 현상은 파일 시스템의 노화(aging)으로 인해 야기되는 가장 큰 문제점으로, 반복되는 파일의 삭제와 생성에 의해 발생한다. 또, 여러 프로세스가 동시에 디스크 블록을 할당받으려 할 때에도 발생한다. 파일 시스템이 단편화 현상을 잘 해결하지 못하면 고유의 블록 할당 정책(block allocation policy)를 효율적으로 수행 할 수 없게 되고, 디스크의 블록 배치 레이아웃(disk layout)을 유지할 수 없다. 이런 이유들로 인해 파일 시스템의 성능이 저하되고, 대부분의 경우 스스로 회복하기 어렵게 된다[1][2][3].

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 저장 되어있는 파일들의 단편화를 제거해주는 프로그램이며, 대부분의 어플리케이션들은 비슷한 방식으로 접근하고 있다. 파일 시스템의 초기 디자인 단계에서 고려하는 경우는 리눅스의 FFS(Fast File System)을 개량한 것을 예로 볼 수 있다. 기존 FFS를 개발한 McKusick은 기존 블록 할당 알고리즘이 단편화 현상에 취약함을 알고 블록 재할당 단계를 포함한 새로운 블록 할당 알고리즘을 기존 FFS에 적용시켰다. 이는 블록 할당 정책에 따라 일단 할당된 블록들을 디스크에 데이터를 쓰고자 할 때 다시 한 번 클러스터링(clustering)할 수 있도록 해주는 알고리즘이다[4][5].

본 논문에서는 파일 시스템의 노화 정도를 지속적이면서 자동적으로 측정할 수 있는 도구를 설계하고 구현하였다. 우리는 EXT2파일 시스템을 기반으로 한 새로운 파일 시스템을 실험용 하드디스크 파티션에 실험도구(실제 서버의 파일 시스템에서 발생한 workload들의 로그를 기록하고 이를 다시 수행시켜 볼 수 있는 도구)를 사용하여 단편화 현상을 인위적으로 생성하였고, 측정하였다. 자동적인 단편화 측정 시스템은 각각의 아이노드가 블록을 할당할 때마다 해당 블록이 연속적인지 아닌지 판별하여 연속적인 블록의 총 개수를 저장한다. 그리고, 블록 그룹과 슈퍼 블록에 대해서도 이러한 동작을 계속해서 수행하며 레이아웃 스코어를 유지한다. 이것은 이전까지 시도된 적이 없었던 기법으로 특정 시기에 파일 시스템의 구조를 파악하여 단편화 정도를 측정하는 기존 방식과 비교할 때 성능의 오버헤드가 적을 것으로 기대된다. 또한 ioctl() 인터페이스의 수정을 통해 아이노드와 블록 그룹, 그리고 슈퍼 블록의 단편화 관련 정보를 얻을 수 있었다.

2. 관련 연구

2.1 레이아웃 스코어링(Layout Scoring)

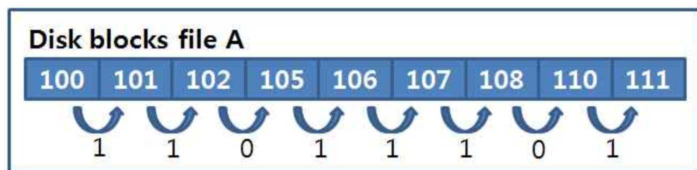
레이아웃 스코어링은 현 파일 시스템의 단편화 정도를 점수로 나타내는 방법이다. [1]에서 제시된 방식으로, 단편화 정도를 0부터 1사이의 점수로 나타내는 방식이다. 그림 1에서와 같이 점수를 측정하는 방식은 하나의 블록이 앞의 블록과 연속하면(같은 파일에 속한 블록이면) 1, 그렇지 않으면 0으로 점수를 할당한 다음 모든 점수를 더하고 (블록 개수 -1)로 나눈다. 이러한 방식을 사용하여 파일 시스템의 현재 단편화 정도를 판단할 수 있다. 이와 관련하여 Keith A. Smith는 각종 파일 시스템의 단편화 정도를 판단하고 실제 파일 시스템에서 변화의

양상을 파악하는 연구[6] 등에 사용하였다. 그리고 [4] 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 두 가지 단점을 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-실제로 모든 파일의 간접블록이 버퍼에 올라와 있을 수는 없을 것이다.- 이 동작은 순간적으로 커다란 I/O의 성능 저하를 가지고 올 수 있다.



< 그림1. 레이아웃 스코어링 방식 >

3. ALS 시스템 설계

3.1 설계목표

- 파일 시스템의 동작 중 자동으로 스코어링을 수행하고 이를 유지함

파일 시스템이 파일에 대해 블록을 할당하거나 해제할 때, 동시에 그에 대한 레이아웃 스코어를 측정한다. 이를 통해 사용자의 조작 없이도 파일 시스템의 노화 정도를 지속적으로 측정할 수 있다. 또 측정된 스코어는 디스크에 저장되어 계속해서 유지할 수 있어야 한다.

- 각 아이노드, 그룹별로 스코어링을 수행함

스코어링을 수행하는 단위는 하나의 파일을 표현하는 아이노드 각각과 근접한 실린더들을 묶어서 구성된 그룹별로 설정한다. 이를 통해 파일 시스템의 단편화에 큰 영향을 주는 아이노드와 그룹을 식별할 수 있고, 단편화 기법을 수행할 대상을 지정하기 위해 활용될 수 있다.

- 성능의 영향을 최소화함

위와 같은 동작을 수행하면서도 성능에의 영향은 최소화한다. 기존 방식은 앞서 설명한대로 파일 시스템의 메타데이터 정보를 읽어 들여야 하는데, 현재 본 보고서에서 제안하는 방식은 이러한 추가적인 동작을 최소화하여 설계한다.

3.2 블록 할당 시 스코어링 기법

우리는 앞서 소개했던 [1] 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 쓰여지는 동안 자동적이면서 계속적으로 측정하기 위해 리눅스 파일 시스템에서 파일이 자라나는 모델을 분석하였다. 리눅스 파일 시스템에서는 크게 두 가지 방식으로 파일에 데이터가 쓰여진다.

첫째, 파일이 생성되고 연속적으로 데이터가 쓰여지는 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 필드를 두 개 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 퍼센티지로 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 블록

의 비율을 계산할 수 있다. 블록 그룹에서의 방식과 마찬가지로 성능에는 문제가 없다.

3.3 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, 앞서 설명했던 SMP 시스템을 고려한 동작에서 할당되었던 블록이 취소되고 다시 할당을 수행할 경우이다.

우리는 단순히 파일이 삭제될 때 해당 아이노드에 기록 되어 있던 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 그러한 경우에 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.

3.4 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 request 필드에 넣어 주어서 정보를 요청할 수 있다.

"EXT2JSLEE_IOC_GET_AGING_COUNT_INODE"는 한 아이노드에 대한 정보를 얻을 수 있다. 이 파라미터를 이용해 ioctl()을 호출하였을 경우, 그림 2와 같이 출력된다. Blocks는 파일이 사용하고 있는 전체 블록을 나타낸다.

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

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

< 그림 2. 아이노드에 대한 정보 >

"EXT2JSLEE_IOC_GET_AGING_COUNT_SUPER"는 전체 블록 그룹들과 슈퍼 블록에 대한 정보를 출력한다. 예를 들어, 노화가 진행된 파일 시스템에 대한 정보는 그림 3과 같다. SCORE는 연속적인 블록을 나타내는 레이아웃 스코어, ALL은 전체 할당된 블록, FREE는 할당되지 않은 블록의 개수를 나타낸다.

3.5 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 정

```
<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
```

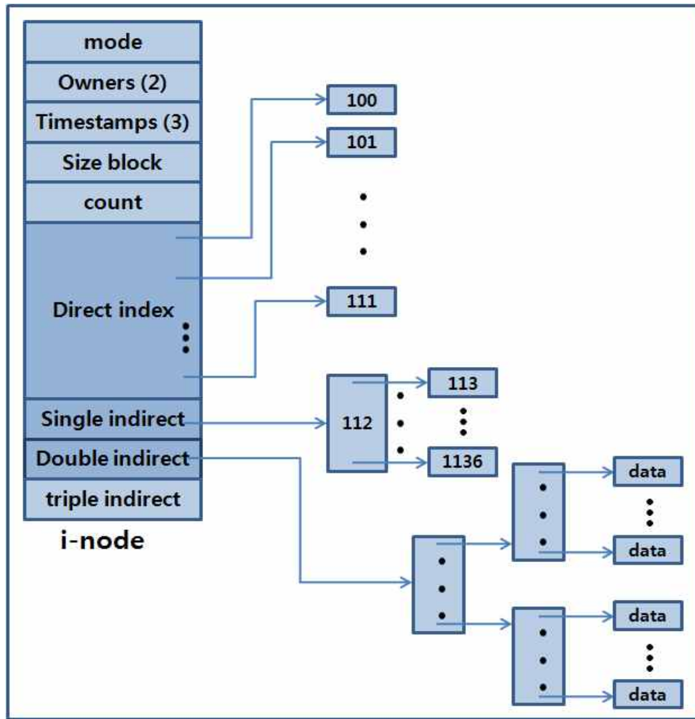
< 그림 3. 블록 그룹과 슈퍼 블록에 대한 정보 >

책에서 문제점을 발생시킨다. 슈퍼블록이나 아이노드 정보가 포함되어있는 그룹 디스크립터 블록 등의 메타데이터들은 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 논문에서 제안하는 액세스의 성능은 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 4에서와 같이 12개의 직접 접근 블록이 블록 번호 0부터 11인 블록에 할당되고, 그 이후에 1단계 간접 접근블록을 사용해야 하는 경우, 12번 블록이 할당되고 그 이후의 데이터 블록은 13번부터 할당이 되어야 이상적이다. 물론 이 12번 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 12번 블록은 13번부터 1036번 블록에 대한 주소를 저장하고 있을 것이다. 그렇다면 12번과 13번은 연속적이지만 1036번 블록에 대해 접근하려고 할 때는 12번과 1036번은 연속적이지 않다. 그러나 우리는 순차적인 읽기가 이루어지는 것을 가정하였기 때문에 13번 데이터 블록에 접근이 되는 순간, 12번 블록의 데이터는 메모리에 캐싱되어 있음을 알 수 있다. 따라서 1036번 블록에의 순차적인 접근은 12번 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.

4. 시스템 구현

4.1 구현 환경

우리는 위에서 설계한 자동적인 레이아웃 스코어링 기법을 리눅스 2.6.18 버전의 EXT2 파일 시스템에 구현하였다. 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.



< 그림 4. I-node 내부 구조 >

4.2 커널 모듈 형태의 구현

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.

4.3 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.

4.4 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.

4.5 관련 구조체의 수정 내용

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 아이노드, 블록그룹, 슈퍼블록의 레이아웃을 정해놓은 구조체인 ext2_inode, ext2_group_desc, ext2_super_block 등이 있다.

본 논문에서는 아이노드의 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.

5. 시험 결과

5.1 실험 환경

앞서 설계하고 구현한 자동적인 레이아웃 스코어링 시스템을 실험해보고 평가해보기 위하여 우리는 아래와 같은 실험 환경을 구성하였다.

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

해당 파티션에 새로운 기능을 추가한 EXT2 파일 시스템으로 마운트하여 사용하였다. 실험 중, 모든 I/O 작업에 대해서는 일반적인 파일 오퍼레이션들을 사용하였으며, C 라이브러리의

FILE 스트림을 사용하지 않았다. 이는 C 라이브러리 내에서의 버퍼링을 이용하기 때문에 성능에 영향을 줄 수 있다.

5.2 동작의 검증

우리는 새로운 파일 시스템을 실제 하드디스크 파티션에 사용하여 단편화 현상을 생성해 보았다. [1]에서 K.A. Smith는 실제 사용되는 서버의 파일 시스템에 발생한 작업 (workload)들의 로그를 기록하고 이를 다시 수행시켜볼 수 있는 실험 도구를 개발하고 제공한다. 이 도구를 사용하면 실제로 파일 시스템에 행해진 파일의 생성과 쓰기, 삭제, 디렉토리 생성 등을 새로운 파티션에서 빠르게 다시 수행 시켜볼 수 있다. 이를 통해 파일 시스템의 노화를 실제로 수행해 볼 수 있고, 단편화 현상의 양상을 관찰할 수 있다.

[1]에서는 NFS(Network File System)으로 연결된 502 MBytes 파티션에 대해 수행된 10개월간의 파일 시스템 워크로드를 수집하였다. 그 결과로 80만개의 파일 시스템 오퍼레이션들이 기록 되었고, 이 오퍼레이션들을 실험용 머신에서 다시 수행시키는 데는 대략 9분 40초 가량 소요된다. 이 워크로드는 처음에 전체 파티션 중 10% 가량을 사용하지만, 실행 중간에는 최대 90% 까지 이용률이 증가하고, 마지막에는 70% 가량을 차지한다. 그리고 대략 48.6GBytes의 데이터를 디스크에 기록한다.

본 논문에서는 이 워크로드를 사용하여 새로운 파일 시스템에 단편화 현상을 인위적으로 발생 시켰다. 실험 대상이 되는 파티션은 1.4GBytes의 용량을 갖고 있는데, 이는 단편화 현상을 보다 심화시키기 위한 것이다.

5.3 실험 결과

첫 번째 실험으로 해당 기능이 정확히 동작하는지 살펴보기 위하여 파일을 생성하는 도중 관련 정보를 출력해 보았다. 그림 5는 71,527 bytes 크기의 파일 하나를 새로운 파일 시스템에서 생성했을 때 커널 메시지를 출력한 것이다. PREV는 이전에 할당된 블록을 뜻하고, ALLOC_BLOCK은 이번 요청을 통해 할당된 블록을 의미한다. temp는 레이아웃 스코어를 계산하기 위한 임시 변수 값이다.

```
[root@localhost ~]# cp examples.tar.gz /home/jslee/test
[root@localhost ~]# dmesg -c
Inode = 14 PREV =  20480 ALLOC_BLOCK =  20480 temp =   0
Inode = 14 PREV =  20480 ALLOC_BLOCK =  20481 temp =   1
Inode = 14 PREV =  20481 ALLOC_BLOCK =  20482 temp =   1
Inode = 14 PREV =  20482 ALLOC_BLOCK =  20483 temp =   1
Inode = 14 PREV =  20483 ALLOC_BLOCK =  20484 temp =   1
Inode = 14 PREV =  20484 ALLOC_BLOCK =  20485 temp =   1
Inode = 14 PREV =  20485 ALLOC_BLOCK =  20486 temp =   1
Inode = 14 PREV =  20486 ALLOC_BLOCK =  20487 temp =   1
Inode = 14 PREV =  20487 ALLOC_BLOCK =  20488 temp =   1
Inode = 14 PREV =  20488 ALLOC_BLOCK =  20489 temp =   1
Inode = 14 PREV =  20489 ALLOC_BLOCK =  20490 temp =   1
Inode = 14 PREV =  20490 ALLOC_BLOCK =  20491 temp =   1
Inode = 14 PREV =  20491 ALLOC_BLOCK =  20492 temp =   1
Inode = 14 PREV =  20492 ALLOC_BLOCK =  20493 temp =   2
Inode = 14 PREV =  20493 ALLOC_BLOCK =  20494 temp =   1
Inode = 14 PREV =  20494 ALLOC_BLOCK =  20495 temp =   1
Inode = 14 PREV =  20495 ALLOC_BLOCK =  20496 temp =   1
Inode = 14 PREV =  20496 ALLOC_BLOCK =  20497 temp =   1
Inode = 14 PREV =  20497 ALLOC_BLOCK =  20498 temp =   1
[root@localhost ~]# █
```

< 그림 5. 연속적으로 블록이 할당된 파일의 예 >

결과를 살펴보면, 우선 첫 번째 할당 요청에 대해서 PREV 값이 20480으로 설정되어 있는 것을 알 수 있다. 이는 앞서 파일이 새롭게 할당되거나, 메모리에서 메타데이터가 해제된 이후 다시 접근할 때, 이전에 할당된 블록의 정보가 유실되는 문제를 ext2_find_goal() 함수를 사용하여 해결한 방식이 정확이 동작하고 있다는 것을 나타낸다. 또한, 이때는 첫 번째 블록의 할당이기 때문에 비연속적인 블록으로 인식하였다. 그 이후에는 계속해서 연속적인 블록들이 할당되고, 레이아웃 스코어가 1씩 합산되는 것을 알 수 있다. 이것은 예약 할당 기능이 성공적으로 수행된 결과라고 생각할 수 있다. 그리고 temp 변수가 2의 값을 가지는 것을 볼 수 있는데, 이것은 첫 번째 1단계 간접 접근 블록이 할당된 결과이다. ext2_alloc_branch() 함수 내에서 필요한 경우 간접 접근 블록의 리스트를 함께 할당하기 때문에 임시 변수 temp가 2를 가리키는 것이다. 20492 블록이 1단계 간접 접근 블록으로 사용되고, 20493 블록은 그 다음의 데이터 블록으로 사용된다.

이 파일을 표현하기 위해 사용된 전체 블록은 19개이고, 이 중 데이터 블록은 18개이다. 한 블록이 4096 byte를 나타낼 수 있기 때문에 현재 파일의 크기인 71,527 byte를 표현하기 위해서는 블록이 17.46개, 즉 18개 필요하다는 것을 알 수 있다. 이 파일은 모든 직접, 간접 블록들이 연속적으로 배치되어 있다.

그림 6은 위와 같은 파일을 또 다시 생성하였을 때, 할당이 연속적으로 이루어지지 않은 경우이다. 파일 시스템을 오랜 시간 사용하여 충분히 단편화를 발생시킨 후 실험하였으며, 그림 6의 상단에서 보이는 경고 메시지는 단편화가 심화되었을 때 출력하도록 만든 메시지이다.

ext2_new_inode()를 사용하여 아이노드를 새로 생성하고자 할 때, 해당 아이노드가 속한 블록 그룹이 85% 이하의 연속을 보인다면 경고 메시지를 출력한다. 현재 1, 5, 6, 7, 8, 10, 11 번의 7개 블록 그룹이 상당히 단편화 되어 있다는 것을 알 수 있다.

이 파일은 2번째 블록을 할당할 때, 이전에 할당된 98879 블록의 바로 뒤인 98880 블록을 얻지 못하고, 98892 블록이 할당되었다. 이로 인해 temp 변수는 0이 설정되었고, 파일의 전체 연속성도 업데이트 되었고, 이후로도 단편화가 계속 진행된 것을 확인 할 수 있다.

```
[root@localhost ~/test/test1]# cp 403 c4
[root@localhost ~/test/test1]# dmesg -c
Warning!! Group No     1 BG Continuity Rate =  83% below 85!!
Warning!! Group No     5 BG Continuity Rate =  76% below 85!!
Warning!! Group No     6 BG Continuity Rate =  80% below 85!!
Warning!! Group No     7 BG Continuity Rate =  67% below 85!!
Warning!! Group No     8 BG Continuity Rate =  82% below 85!!
Warning!! Group No    10 BG Continuity Rate =  84% below 85!!
Warning!! Group No    11 BG Continuity Rate =  84% below 85!!
Inode = 76005 PREV = 180224 ALLOC_BLOCK =  98878 temp =   0
Inode = 76005 PREV =  98878 ALLOC_BLOCK =  98879 temp =   1
Inode = 76005 PREV =  98879 ALLOC_BLOCK =  98892 temp =   0
Inode = 76005 PREV =  98892 ALLOC_BLOCK =  98893 temp =   1
Inode = 76005 PREV =  98893 ALLOC_BLOCK =  98894 temp =   1
Inode = 76005 PREV =  98894 ALLOC_BLOCK =  98895 temp =   1
Inode = 76005 PREV =  98895 ALLOC_BLOCK =  98896 temp =   1
Inode = 76005 PREV =  98896 ALLOC_BLOCK =  99223 temp =   0
Inode = 76005 PREV =  99223 ALLOC_BLOCK =  99267 temp =   0
Inode = 76005 PREV =  99267 ALLOC_BLOCK =  99268 temp =   1
Inode = 76005 PREV =  99268 ALLOC_BLOCK =  99269 temp =   1
Inode = 76005 PREV =  99269 ALLOC_BLOCK =  99270 temp =   1
Inode = 76005 PREV =  99270 ALLOC_BLOCK =  99287 temp =   0
Inode = 76005 PREV =  99287 ALLOC_BLOCK =  99778 temp =   0
Inode = 76005 PREV =  99778 ALLOC_BLOCK =  99840 temp =   0
```

< 그림 6. 비연속적으로 파일이 할당된 파일의 예 >

6. 결론 및 향후 연구과제

본 논문에서는 리눅스에서 일반적으로 사용되는 EXT2 파일 시스템의 노화 정도를 측정하기 위해 자동적이고 지속적인 단편화 측정을 위한 레이아웃 스코어링(Layout Scoring)기법을 적용하여 시스템을 설계 및 구현 하였다. 또한 이 시스템을 이용하여 실제 하드디스크 파티션에서 단편화 현상을 생성해 보

고, 파일 시스템에서 발생하는 단편화 정도를 측정해 보았다. 자동적인 단편화 측정 시스템은 각각의 아이노드가 블록을 할당할 때마다 해당 블록이 연속적인지 아닌지 판별하여 연속적인 블록의 총 개수를 저장한다. 그리고, 블록 그룹과 슈퍼 블록에 대해서도 이러한 동작을 계속해서 수행하며 레이아웃 스코어를 유지한다. 그리고 새롭게 구현한 EXT2 파일 시스템을 모듈화하여 기존 시스템의 패치나 재컴파일, 재시작 없이 사용할 수 있도록 하였다.

향후 연구과제로는 위와 같은 실험 결과를 바탕으로 레이아웃 스코어링 기법을 통해 측정된 파일 시스템의 단편화 정도를 등급으로 나누어 차등적인 단편화 해소가 이루어 질 수 있는 정책을 세워야 하며, 정책에 맞는 단편화 해소 시스템의 개발이 요구된다.

7. 참고문헌

[1] K.A.Smith and M. Seltzer,"File system aging: increasing the relevance of file system benchmarks," ACM SIGMETRICS Performance Evaluation Review, vol. 25, no. 1, 1997, pp. 203-213.

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

[3] K.A. Smith, "Workload-Specific File System Benchmarks," Harvard University, 2001

[4] K.A.Smith and M. Seltzer, "A comparison of FFS disk allocation policies," Processing of the Annual Technical Conference on USENIX 1996 Annual Technical Conference table of centents, 1996, pp. 2-2.

[5] M.K. McKusick, et al., "A Fast File System for UNIX," ACM Transactions on Computer Systems, vol. 2, no. 3, 1984, pp. 181-197

[6] K.A.Smith and M. Seltzer, "File Layout and File System Performance," Harvard University Computer Science Technical Report TR-35-94. December, 1994