저작자표시 **2.0** 대한민국

이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.
- 이차적 저작물을 작성할 수 있습니다.
- 이 저작물을 영리 목적으로 이용할 수 있습니다.

다음과 같은 조건을 따라야 합니다:

저작자표시. 귀하는 원저작자를 표시하여야 합니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 이용허락규약(Legal Code)을 이해하기 쉽게 요약한 것입니다.

Disclaimer

공학석사학위논문

멀티 쓰레드 환경을 위한 Large I/O 파일 시스템

Large I/O File System for Multithread Environment

2013년 8월

서울대학교 대학원

전기·컴퓨터 공학부

최 윤 희

멀티 쓰레드 환경을 위한
Large I/O 파일 시스템

Large I/O File System
for Multithread Environment

지도교수 엄 현 상

이 논문을 공학석사학위논문으로 제출함

2013년 8월

서울대학교 대학원
전기·컴퓨터 공학부
최 윤 희

최윤희의 석사학위논문을 인준함

2013년 8월

위 원 장 ______ 한 상 영 ______ (인)
부 위 원 장 ______ 엄 현 상 ______ (인)
위 원 ______ 이 광 근 ______ (인)

초록

오늘날 대용량 데이터를 처리하는 클러스터의 경우 저장 공간으로 플래시 메모리(SSD)로 쓰기엔 성능에 비해 고비용이기 때문에 일반적으로 하드디스크를 사용한다. 하드디스크는 랜덤 접근이 바로 되는 플래시메모리와는 달리 랜덤한 작업보다는 순차적인 작업을 해야 효율성이 높아진다.

하지만 대용량 데이터를 처리할 때 일반적으로 기존의 OS 디스크 스케줄러와 파일시스템(Linux, ext2 등)에 의존하게 된다는 문제점이 있다. 즉, 기반의 운영체제가 디스크 할당이나 디스크 스케줄링을 담당하게 되어 단편화나 여러 클라이언트들 사이에서 디스크를 공유하게 되는 문제를 겪게 된다. 그래서 본래라면 순차적으로 처리되었을 쓰기 작업이 여러 클라이언트들이 디스크를 공유하게 되면서 대역폭이 줄어드는 현상이 일어난다. 이러한 현상은 보통의 대용량 데이터 처리를 순차적으로 한다는 점, 병렬로 한다는 점을 고려하면 문제가 될 수 있으며 이 논문에서는 여기에서 디스크에 직접 적는 크기를 처음부터 크게 잡아주어 내려주는 방식을 제안하며, 이를 위해 멀티 쓰레드 환경에서 sequential write에 대한 디스크의 효율을 살릴 수 있는 프로토타입 파일 시스템을 디자인하고 구현하였다.

주요어 : File System, Large Sequantial Write, Disk Allocation, Free Space Management, Multithread
학 번 : 2011-20945

목차

초록 iii

목차 v

표 목차 vii

그림 목차 viii

제 1 장 서론 1

제 2 장 배경 3

2.1 멀티스레드 환경 3

2.2 Raw Disk 성능 4

2.3 파일 시스템 5

2.4 관련 연구 7

제 3 장 파일 시스템 디자인 11

3.1 기본 구조 11

3.2 디스크 할당 14

3.3 빈 공간 관리 17

제 4 장 Implementation Details 19

4.1 Low-level Function 19

4.2 High-level Function 21

4.3 Implementation Issue 22

제 5 장 및 검증 **24**

5.1 Sequential Write Test 24

5.2 Benchmark Test 25

제 6 장 결론 26

참고 문헌 27

ABSTRACT 29

표 목차

표 5.1 sequential write test time & throughput 24

그림 목차

그림 2.1 write throughput per block size 4

그림 2.2 리눅스 구조 5

그림 2.3 커널 시스템 콜 스택 6

그림 3.1 파일 시스템 디자인(Linked Allocation) 11

그림 3.2 Super Block 구조 12

그림 3.3 dentry 구조 13

그림 3.4 파일 핸들 구조 13

3.5 Linked Allocation 파일 구성 14

그림 3.6 Blocklink Table 15

그림 3.7 Contiguous Allocation 파일 구성 15

그림 3.8 CA+append 파일 구성 17

그림 3.9 Contiguous Allocation 파일시스템 전체구조 18

그림 3.10 Contiguous Allocation Block Bitmap 구조 18

그림 4.1 파일 열기 도식 21

그림 4.2 파일 읽기 도식 22

그림 4.3 파일 쓰기 도식 22

그림 5.1(a) fio write 그림 5.1(b) fio read 26

그림 5.2(a) iozone write 그림 5.2(b) iozone write 26

제 1 장 서론

오늘날 대용량 데이터를 처리하는 클러스터의 경우 저장 공간으로 플래시 메모리(SSD)로 쓰기엔 성능에 비해 고비용이기 때문에 일반적으로 하드디스크를 사용한다. 하드디스크는 랜덤 접근이 바로 되는 플래시메모리와는 달리 랜덤한 작업보다는 순차적인 작업을 해야 효율성이 높아진다. 데이터를 순차적으로 접근하는 것이 랜덤하게 접근하는 것 보다 더 빠른 이유는 하드디스크가 일하는 방식이 기계적이기 때문이다. 디스크 헤드가 요청된 데이터를 접근할 수 있는 적합한 디스크 실린더를 찾는 seek 오퍼레이션에 의해 I/O 처리에 훨씬 더 많은 시간이 걸린다. 랜덤하게 읽는 작업이 순차적으로 읽는 작업보다 seek 오퍼레이션을 많이 하기 때문에 작업률이 저하된다. 멀티쓰레드 환경에서 기반의 운영체제가 디스크 할당이나 디스크 스케줄링을 담당하게 되어 단편화나 여러 클라이언트들 사이에서 디스크를 공유하게 되는 문제를 겪게 된다. 본래라면 순차적으로 처리되었을 쓰기 작업이 여러 클라이언트들이 디스크를 공유하게 되면서 대역폭이 줄어드는 현상이 일어난다. 순차적으로 실행되는 평균 길이가 줄어들면서 랜덤한 작업이 일어나게 되고 결국 디스크 드라이브를 효율이 저하된다. 이러한 현상은 보통의 대용량 데이터 처리를 순차적으로 한다는 점, 병렬로 한다는 점을 고려하면 문제가 될 수 있으며 이러한 현상을 줄이기 위해서는 OS로 가는 요청을 처리하기 위한 순서대로 표시해 디스크 별로 메모리에 버퍼를 두어 스케줄링 하는 방식이 있다.[5] 이 논문에서는 여기에서 디스크에 직접 적는 크기를 처음부터 크게 잡아주어 내려주는 방식을 제안하며, 이를 위해 쓰고 읽는 기능을 하는 파일시스템을 디자인하고 구현하였다.

이 논문에서는 이 목적에 맞춰 섹션 2에서는 먼저 멀티스레드 환경과 Raw Disk 성능에 대해서 이야기하고 파일시스템 기능을 디자인하기 전 기존 파일 시스템은 어떠한 구조로 되어있는지 살펴본다. 섹션 3에서는 데이터 구조 및 디스크 할당 방식과 빈 공간 관리 등 파일시스템 디자인이 맞춰 어떠한 구조와 방식이 필요한지 살펴보며 섹션 4에서는 이를 위해 구현해야 했던 함수들 및 이슈에 대해 설명한다. 섹션 5는 구현한 파일시스템이 목적에 맞게 구현이 되었는지 실험 및 검증을 하였다. 제 6장 에서는 요약 및 결론을 통한 정리를 진행한다.

제 2 장 배경

2.1 멀티스레드 환경

예전에는 고사양이라 하면 frequency가 높은 CPU를 의미했지만 오늘날의 고사양이란 CPU 여러 개 즉, 멀티코어를 의미한다. 요 근래에는 가정에서 쓰이는 보급형 PC도 대부분은 Dual이나 Triple 또는 Quad core CPU가 기본이다. 멀티코어는 겉으로는 하나의 CPU로 보이지만 내부에는 복수 개의 CPU가 들어있는 형태이다. 과거에는 멀티코어(CMP)가 고가였지만 지금은 저가 CPU 조차 멀티코어다. 멀티코어 프로세서는 보다 강력한 성능과 소비 전력 절감, 그리고 여러 개의 작업을 보다 효율적으로 한 번에 처리하기 위해 두 개 이상의 프로세서가 붙어있는 집적회로를 가리킨다. 듀얼코어는 한 컴퓨터 내에 두 개 이상의 독립된 프로세서가 설치된 경우와 종종 비교되곤 하는데, 듀얼코어의 경우 두 개의 프로세서가 실제로는 하나의 소켓을 통해 꽂혀지기 때문에 연결이 더 빠르다. 이론적으로는 듀얼코어 프로세서가 싱글코어 프로세서보다 두 배나 더 강력해야 하지만, 실제로는 듀얼코어 프로세서가 싱글코어 프로세서보다 약 1.5배 정도 더 강력하기 때문에, 약 50% 정도의 성능 향상만을 기대할 수 있는 것으로 알려지고 있다. 현재, 싱글코어 프로세서가 복잡도나 속도 측면에 있어 거의 물리적 한계에 도달하고 있기 때문에, 멀티코어 프로세서와 관련된 산업이 점차 성장하고 있는 추세이다. 이러한 하드웨어적 패러다임의 변화는 소프트웨어 개발에도 영향을 주어 하나의 프로그램에서 이전과는 달리 여러 스레드들이 한 머신에서 서로 간섭하지 않고 조화롭게 실행되는 구조로 성능향상을 꾀하고 있다. CPU는 동시에 한 개씩의 스레드만을 실행시킬 수 있다. 만약 스레드가 여러개가

생성되면 CPU는 각각의 스레드를 시분할하여 번갈아가면서 실행하게 되며, 이전 스레드의 문맥 정보인 레지스터값, 실행 중인 스택 정보 등을 백업받고 백업받아 놓았던 다음 스레드의 문맥정보를 로딩하는 과정을 거치게 된다. 이 과정을 문맥 교환이라고 하는데, 이러한 스레드가 많아질수록 문맥 교환에 많은 부하가 발생하기 때문에 성능이 저하된다. 이 논문에서는 문맥 교환에서의 부하가 아닌 여러 스레드가 디스크를 공유하면서 쓰는 데에 있어서의 성능 저하를 관찰하여 블록 사이즈에 따라 성능 저하에 차이가 있음을 발견하였고 이를 기반으로 삼아 멀티 스레드 환경에서 순차적으로 쓰기 작업을 동시에 할 시에도 성능저하가 적은 파일 시스템 기능을 구현하였다.

2.2 Raw Disk 성능

멀티 쓰레드 환경에서 데이터 블록이 작을 때 성능의 저하가 있다. 아

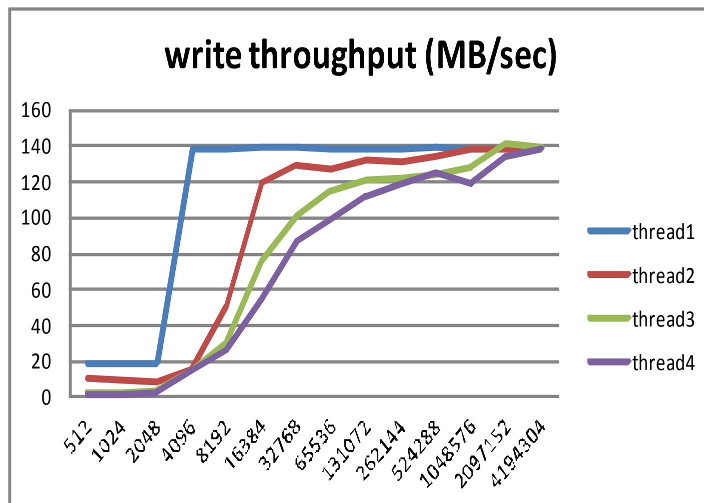

그림 2.1 write throughput per block size

래의 그래프를 보면 쓰기 작업을 하는 스레드가 한 개 일 때보다 스레드 코어 수만큼 늘어날수록 쓰기 작업률에 대한 블록 사이즈의 영향이 더 커짐을 알 수 있다. 즉, 데이터 블록 사이즈를 크게 해주면 멀티 쓰레드 환경에서의 성능 저하를 없앨 수 있을 것이다.

2.3 파일 시스템

응용프로그램은 시스템 콜 인터페이스를 통해 하드웨어를 조작을 wrapping 하여 사용할 수 있게 된다. 시스템 콜은 리눅스 커널을 통해 하드웨어를 직접 조작한다. 파일시스템도 디스크를 조작하기 위해 결국

그림 2.2 리눅스 구조

커널을 거치게 된다. 즉, 파일시스템은 리눅스에서는 커널에 포함된 영역이며 기억 장치 상의 데이터를 파일이라고 하는 형식을 통해 액세스 순서를 일관성 있게 제공하며 파일을 정의하고 구성하며 데이터 엑세스 관리를 한다. 커널에서는 파일을 바이트 스트림으로 보게 된다. 바이트 스

트림 형태로 된 파일들을 커널을 통해 사용자가 조작할 수 있도록 해주는 것이 파일 시스템이다.

일반적으로 파일 시스템은 파일에 inode로 불리는 데이터 구조를 할당하여 관리하며 각 inode는 그 파일의 속성이나 파일의 실제 데이터가 있는 2차 기억장치 상의 블록 번호 등을 관리한다. 모든 파일은 루트에서 시작하여 1개의 트리 상태로 관리하며 파일위치(경로명) 이용해 파일을 지정할 수 있다. 또한 사용자가 파일시스템에 접근하게 될 시에 커널에서 바로 접근하는 것이 아니라 가상 파일 시스템이라는 공통의 인터페이

그림 2.3 커널 시스템 콜 스택

스를 거치게 된다. 가상 파일 시스템은 사용자로부터 어떤 이벤트가 들어오면 여러 종류의 파일시스템에 대해 공통으로 처리되는 부분을 담당하며, 그 아래 다양한 종류의 파일시스템을 배치 가능하다. 데이터가 로컬 디스크 상에 있는지 네트워크 상에 있는지에 따라 각각의 파일시스템마다 다른 부분은 은폐하고 파일을 추상화하여 취급한다.

2.4 관련 연구

원래의 운영체제가 디스크 할당이나 디스크 스케줄링을 담당하게 되어 단편화나 여러 클라이언트들 사이에서 디스크를 공유하게 되는 문제를 겪게 된다. 첫 번째 문제는 디스크 스케줄링에서 디스크를 공유하는 작업에 의한 영향이다. 한 노드당 1개에서 4개의 Writer를 두었을 때의 쓰기 대역폭 성능을 측정한 결과, 하나의 Writer에서 2개로 늘었을 때 대역폭이 38%정도 줄어듦을 확인할 수 있었고, Reader를 두었을 때도 비슷한 결과를 나타냈다. 즉, 동시에 쓰기 작업을 많이 할수록 대역폭은 감소하고 순차적으로 실행되는 평균 길이가 현저히 작아진다. (4MB에서 200KB이하로) 원인은 디스크의 랜덤 seek 가 자주 일어나서 디스크 헤드가 비효율적으로 움직이기 때문이다. 이러한 현상은 동시 접근이 보통인 환경에서 문제가 된다. 또한 여러 Writer가 쓰기 작업을 했을 경우 단편화 문제가 커진다. 또한, 읽기 작업을 할 때의 읽기 대역폭이 하나의 writer가 작업을 했을 경우 가장 좋다. 이는 기존 파일시스템의 배치 정책이 일반적으로 보통 작은 단위(128KB)로 유지되기 때문이다.

이러한 디스크 스케줄링, 단편화에 대한 문제 해결을 위해 HDFS 레벨의 스케줄링을 한 관련 연구가 있다.[5] Hadoop[1]과 같은 대용량 데이터를 처리하는 프레임워크는 일반적으로 MapReduce[2]를 수행할 시에 입력 데이터를 스캔하여 처리하게 된다. 그래서 앞서 말한 디스크의 특성에 맞게 Hadoop 프레임워크의 Hadoop Distributed File System(HDFS)[3]도 seek 수를 줄이기 위해 큰 블럭(64MB)과 스트리밍 접근 패턴을 사용하고 있다. 하지만 HDFS는 결국엔 기존의 OS 디스크 스케줄러와 파일시스템(Linux, ext2[4] 등)에 의존하게 된다는 문제점이 있다. 이를 해결하기 위해 OS로 가는 요청을 처리하기 원하는 순서대로

나타내고, 각 디스크 별로 메모리에 버퍼를 둬서 요청을 기다리게 하여 이를 64MB의 큰 단위로 스케줄링한다. 이렇게 되면 OS의 관점에서 각 디스크 당 하나의 클라이언트가 접근하게 된다. 동시 쓰기 작업이 증가하게 되더라도 각각의 디스크에 할당된 thread가 데이터를 큰 단위로 내려주기 때문에 순차적으로 접근할 수 있도록 한다. 평균적으로 순차적으로 실행되는 용량은 1개~4개의 쓰기 작업 모두 6MB 정도로 비슷한 수치를 보였으며 쓰기 대역폭도 70~80MB로 기존의 30MB~60MB보다 향상되었다.

스토리지 시스템을 공유하여 쓰는 환경에서도 서로 다른 워크로드들 사이에서의 간섭으로 인해 디스크 효율성이 저하된다.[6] 디스크 효율성은 공간적 지역성(spatial locality)의 영향을 많이 받게 되는데 같은 어플리케이션에서의 요청들(intra-application request)이 다른 어플리케이션에서(inter-application request) 보다 더 강한 지역성을 띄게 된다. 그래서 I/O 요청 스케줄러에서 지역성을 높이기 위해서는 다른 어플리케이션들과 batch 처리하기 이전에 하나의 어플리케이션에서의 요청을 계속적으로 dispatch해야 디스크의 탐색 시간을 최소화시킬 수 있다. I/O 스케줄링을 하는 관점에서는 I/O 요청을 처리할 때 QoS 요구 순서와는 다르게 디스크 효율성을 고려한 순서에 따라 서비스하게 되는데, 만약 QoS 요구 순서에 맞게 처리하게 되면 스토리지 시스템이 요청을 처리할 때 효율성은 저하된다.

intelliQos[6]는 사용자가 관찰할 수 있는 QoS만을 시스템에서 채워주도록 하고 스토리지 시스템에서 사용자에게 보고되지 않을 동안의 QoS 요구는 잠시 쉬게 하는 방식을 제안하였다. 이 휴식시간 (relaxation)은 I/O 스케줄러가 디스크 효율성을 향상시키는데 큰 역할을 한다. 구현한 프로토타입 시스템에서 스케줄러는 사용자가 어플리케이션의 출력에서

성능 저하를 느끼지 않는 선에서 원하는 순서대로 요청을 처리하도록 한다. 이를 위해 컨테이너라는 개념을 이용해 QoS를 만족하면서도 디스트 효율성을 향상시키는 intelliQoS 시스템을 디자인하고 구현하였다.

일어나는 출력(어플리케이션 서버로부터 서비스 구독자에게로 가는 메시지)과 이 출력에 연관되는 요청이 컨테이너를 구성한다. 다시 말해서, 컨테이너는 해당 출력을 위해 완료되어야 하는 I/O 요청들로 구성된다. 보통 한 프로세스에서 나오는 동기화된 요청이 같은 프로세스에서 즉시 따라 나오는 출력과 연관되어 컨테이너를 구성한다. OS 커널에서는 요청과 출력을 관찰하여 관계를 파악할 수 있다. 만약 어플리케이션 서버와 스토리지 서버가 분산되어 있으면 컨테이너를 식별하는 정보를 어플리케이션 서버로부터 스토리지 서버로 보내야 할 것이다. 어플리케이션 서버가 가상머신이고 같은 호스트에 스토리지 서버가 함께 존재하는 경우라면 하이퍼바이저나 호스트 VM이 스토리지로 가는 어떤 요청이나 NIC를 통해 클라이언트로 가는 출력 메시지를 모니터할 수 있다. 게스트 OS의 도움 없이도 guest VM 단위로 출력과 요청을 연관 지을 수 있다. 그렇게 VM의 출력 발생에 따라 같은 게스트 VM의 요청들을 컨테이너들로 구성한다.

또한 일반적으로 각 요청에 deadline이 부여되는 기존의 QoS 요구는 각각의 요청이 deadline을 지키면 만족한다고 가정하게 되지만 이 논문에서는 사용자에게 관찰되는 출력을 QoS 요구기준으로 한다. 컨테이너 안에서는 사용자에게 관찰되는 출력이 없기 때문에 이론적으로 컨테이너 안의 모든 요청들은 다음 출력이 나타나기 전까지만 요청들을 처리하면 사용자에게 보이는 QoS를 맞출 수 있다. 그 때문에 컨테이너의 마지막 요청의 deadline까지 deadline이 확장될 수 있다. 즉, 다음 출력이 일어나야 하는 때가 해당하는 컨테이너의 deadline이 되며, 이 컨테이너의

deadline만 맞추게 된다면 컨테이너 안에서 각각의 개별적인 요청들은 사용자가 관찰할 수 있는 QoS를 지키는데 큰 영향을 끼치지 않는다.

컨테이너 크기가 커지면 지역성이 더 강해져서 디스크 효율성을 높일 수 있으나 QoS 요구를 맞추기 어려울 수 있고, 크기가 작아지게 되면 QoS 요구는 확실히 맞출 수 있으나 디스크 효율성이 감소할 수 있다. 이처럼 컨테이너의 크기에 따라 디스크 효율성(locality)와 QoS 간의 tradeoff가 존재한다.

사용자는 디스크의 효율성을 위해 개별 요청의 latency는 늘어나더라도 스토리지 시스템의 최소 throughput(QoS 요구)는 지켜질 수 있다. 그림 1에서 보면 16개 요청의 batching 처리는 QoS requirement를 지키지 못하는 반면에 intelliQoS는 QoS 요구를 잘 지킨다.

위와 같이 제안한 방식으로 구현한 프로토타입 시스템에서 외부 QoS 개런티를 사용자에게 보장하면서도 시스템 효율성(locality)을 80%까지 향상시켰다.

제 3 장 파일 시스템 디자인

3.1 기본 구조

앞에서 언급한 바와 같이 기억장치 상의 데이터를 파일이라고 하는 형식을 통해 접근하도록 하기 위한 파일시스템의 기능을 구현하기 위해서는 메타데이터 영역과 데이터 영역의 구분이 필요하다. 메타데이터 영역은 데이터 영역의 비어있는 공간과 파일을 구성하는 블록과의 연결관계를 관리한다. 다음 [그림3.1]은 파일 시스템 디자인의 기본 뼈대이다. super block부터 block linke table까지 메타데이터의 영역, 그 이후부터

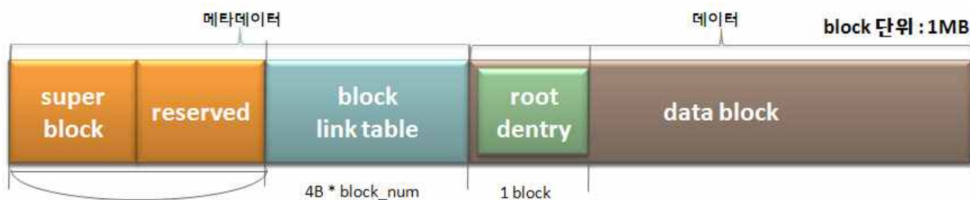

그림 3.1 파일 시스템 디자인(Linked Allocation)

가 데이터의 영역이다. Super Block 영역은 전반적인 파일시스템 정보를 지니고 있으며 예약된 영역은 향후 확장을 위해 위치시켰다. 블록 링크 테이블 영역은 데이터 영역에 있는 블록 사이에 연결 정보를 저장하는 영역이다. 블록 링크 테이블 영역의 크기는 데이터 영역의 블록 개수에 따라 좌우되며, 최소 한 섹터 이상 할당해야 한다. 데이터 영역은 실제 데이터가 저장되는 영역이며, 다시 파일과 디렉터리로 구분된다. 파일은 데이터를 보관하는 실질적인 공간으로 디렉터리는 파일이나 디렉터리 목록을 관리한다. 파일과 디렉터리는 파일시스템의 최소 단위인 블록 단위로 처리한다. 우리가 구현할 파일시스템은 멀티 스레드 환경에서의 디스크 성능 저하를 줄이기 위해 1MB를 묶어서 하나의 블록으로 사용할 것

이다. 또한 디렉터리 구조와 경로 처리의 복잡함을 줄이기 위해 트리 형태의 계층적은 구조 대신 수평적인 구조로 설계하였다. 이를 기반으로 설계한 디자인은 크게 세 가지로 나눠질 것이며 섹션 3.2와 섹션 3.3 에서 설명할 것이다. 현재 섹션 3.1 에서는 기본 데이터 구조에 대해 자세히 설명하겠다.

3.1.1 Super Block 구조체

전체 파일 시스템의 정보를 알아내기 위해서는 파일 시스템 메타데이터 영역에 Super Block 구조체가 존재해야한다. 파일 시스템을 표시하고 관리 정보 등을 다룬다. 메타데이터 시작 위치나 데이터의 시작 위치 또는 파일시스템의 전체 볼륨이나 메타데이터를 다루는데 있어서 필요한 정보 등을 저장할 수 있다. s_blocklink_area_start는 blocklink table의 디스크 오프셋을 저장하고 s_blocklink_area_size는 해당하는 영역의 크기를 저장한다. 해당 영역의 크기는 디스크의 사이즈와 데이터 블록의 개수에 따라 결정된다. s_data_block_area_start도 마찬가지로 데이터 블록의 디스크 오프셋을 저장하며 전체 데이터 블록이 얼마나 들어갈 수 있는지도 s_total_block_count로 나타내었다.

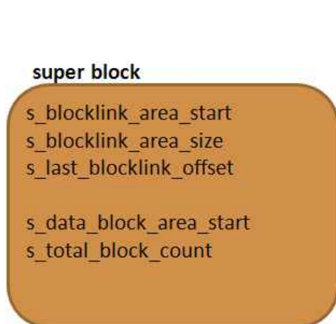

그림 3.2 Super Block 구조

3.1.2 Dentry 구조체

우리 구조에서는 Inode가 없고 Dentry만 존재한다. 결국엔 Dentry 구조

체가 Inode의 역할과 Dentry의 역할을 통합한다. 파일 이름과 파일의 모든 정보를 저장하게 되는데 디렉토리 구조는 하나의 깊이만 갖도록 하여서 루트 디렉토리 만이 데이터 블록의 첫 번째 자리에 위치하게 된다. 파일이 있는지 보거나 읽기 위해서 루트 디렉토리를 봐야하며 Dentry의 엔트리를 통해 파일의 시작 위치를 알 수 있다.

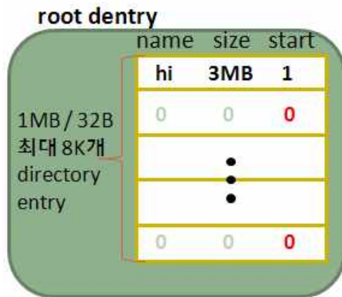

그림 3.3 dentry 구조

3.1.3 File Handle 구조체

File Handle은 파일에 관련된 자료구조 이다. 파일을 블록 단위가 아닌 바이트 단위로 처리하려면 현재 작업이 수행되는 위치에 따라 블록을 이동하는 부분이 필요하다. 이를 파일 포인터라 하며 파일포인터에 따라 블록을 이동시키려면 파일 오프셋을 블록 크기로 나누어 몇 번째 블록인지를 판단해야 한다. 그 후 파일 포인터가 현재 블록 내부에 있는지 확인해야 하며, 다른 블록으로 이동해야 한다면 블록을 탐색해서 찾아야 한다. 이러한 작업을 원만히 처리하려면 파일 포인터의 현재 위치와 파일의 시작 블록의 인덱스, 파일 포인터가 위치한 현재 블록의 인덱스가 필요하다. 그리고 파일의 크기는 전체 블록이 차지하는 크기보다 작을 수 있으므로, 파일 끝을 정확히 판단하려면 파일의 크기도 필요하다. 디렉터리 엔트리에 있는 파일 크기는 파일이 커질 때마다 갱신한다. 그럴 때 루트 디렉터리에서 파일명으로 디렉

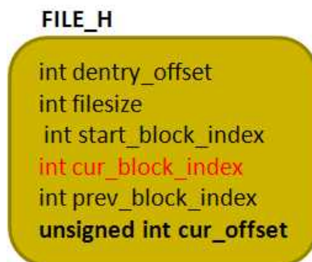

그림 3.4 파일 핸들 구조

터리 엔트를 검색하는 것보다 디렉터리 엔트리의 오프셋을 저장했다가 해당 오프셋만 갱신하는 것이 효율적이기 때문에 디렉터리 오프셋도 포함한다.

3.2 디스크 할당

디스크 공간을 할당하는 방법에는 세 가지가 있다. Linked Allocation, Contiguous Allocation, Indexed Allocation으로 이 논문에서는 Linked Allocation과 Contiguous Allocation을 기반으로 하여 파일시스템의 디스크 할당을 디자인하였으며 이에 대해 현 섹션에서 설명하도록 하겠다.

3.2.1 Linked Allocation

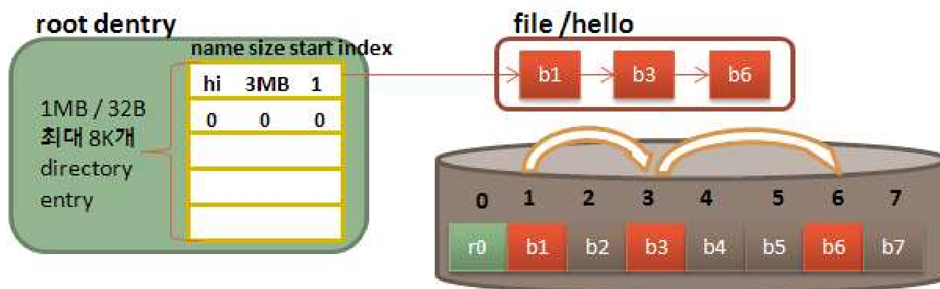

그림 3.5 Linked Allocation 파일 구성

Linked Allocation은 앞서 설명한 파일 시스템의 기본 구조를 기반으로 디자인 되었다. 기존 방식으로는 디스크 블록 단위로 링크를 형성하여 블록의 위치에 영향을 받지 않고 디렉터리는 파일의 시작 번호만을 가지게 되는데, 블록의 영역에 링크를 둔 것이 아니고 우리는 메타데이터 영역에 링크 정보를 놓아두어 블록 링크 테이블이라 명명하였다. 이 디스크 할당 방식은 외부 단편화가 없으며 필요시 빈 블록을 할당하면 된다.

링크 정보를 위해 디스크 공간이 추가로 필요하지만 블록을 1MB로 크게 하여 블록 크기로 디스크 작업이 이루어지기 때문에 필요한 영역이 기존보다 적다. 또 파일의 구성을 위해 [그림3.5] 와 같이 dentry가 파일 데이터 블록의 시작 인덱스를 지니고 있으며 [그림3.6]와 같은 블록 링크 테이블을 통해 현 블록의 다음 블록을 찾아 갈 수 있도록 하였다. 즉, 중간의 블록을 읽기 위해서는 링크 추적이 필요하다. 하지만 블록을 최대한 가까이 두게 된다면 링크의 위치를 메타데이터로써 메모리에 저장하여 활용할 수 있을 것이다. 다음 인덱스가 없는 파일의 마지막 블록은 해당하는 플래그(0xFFFFFFFF)을 두어 구분하였다.

그림 3.6 Blocklink Table

3.2.2 Contiguous Allocation

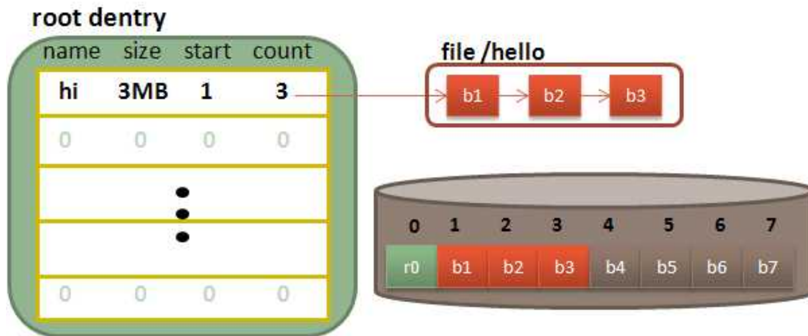

그림 3.7 Contiguous Allocation 파일 구성

Contiguous Allocation은 말 그대로 [그림3.7]와 같이 파일을 디스크 주소에 따라 선형적으로 배열한다. 하나 파일의 시작 주소와 길이를 알면

디스크 공간 계산이 가능하다. 또한 파일의 시작주소와 블록의 오프셋만 으로 데이터 블록을 찾을 수 있기 때문에 블록 접근이 쉽기 때문에 순차적 접근과 direct 접근이 가능하다. 하지만 연속적으로 할당하기 위해서 파일 생성 시 디스크의 연속된 빈 공간을 찾는데 있어서 빈 공간 관리가 필요하다. 이 부분은 섹션 3.3 에서 나올 빈 공간 관리를 통해 디자인을 더 설명하도록 하겠다. 디스크 공간의 크기를 결정하는 데 있어서도 적게 할당 하게 되면 파일 확장에 어려움이 있다. 만약 공간을 적게 할당하여 공간이 부족하게 되면 에러 메시지를 출력하고 프로그램 수행을 종료시키고 더 큰 공간을 출력하고 프로그램 수행을 종료시키는 방법과 새로운 더 큰 연속된 공간을 찾아서 파일 복사 후 이전 파일 삭제한 다음 프로그램 수행을 계속 하는 방법이 있다. 하지만 두 가지 모두 프로그램을 수행하는데 있어서 부하가 커지게 되며 첫 번째 방식은 필요 이상의 공간을 할당해서 결과적으로 디스크 공간 낭비 및 내부 단편화를 발생시킨다. 이 부분에 대해선 파일 확장이 되도록 디자인 한 섹션 3.2.3에서 해결하였다.

3.2.3 Contiguous Allocation + Append

앞 서 설명한 연속적 할당이 파일의 확장에 있어서 취약점이 있었기 때문에 [그림3.8] 와 같이 dentry 엔트리를 디자인하여 파일의 확장이 가능하도록 했다. 기본적으로 연속적인 할당을 하되 파일의 끝에서 적어야 할 경우에는 다른 연속된 공간을 할당 하는 방식으로 디자인 하였으며, 기존의 연속 할당 디자인에서의 파일 확장문제를 해결하여준다. 단점은 마찬가지로 빈 연속된 공간을 관리하여야 하고 만약 충분한 연속된 공간이 없다면 파일을 할당하거나 확장할 수 가 없으며, 데이터 영역에 저장

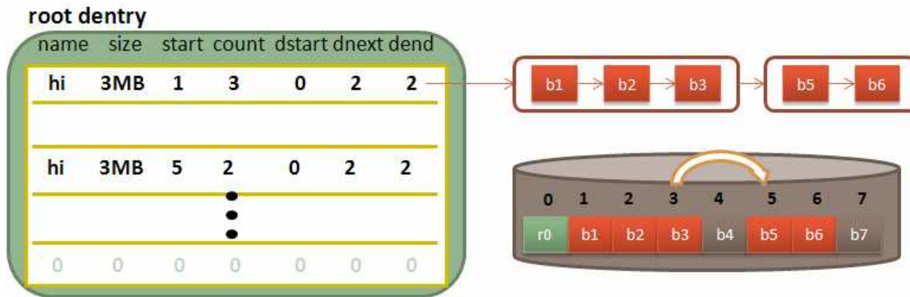

그림 3.8 CA+append 파일 구성

되는 루트 디렉터리의 단층 구조로는 dentry의 개수에 한계가 있기 때문에 파일의 확장이 무한대로 이루어질 수는 없다.

3.3 빈 공간 관리

마지막으로 디스크 할당을 위한 빈 공간 관리에 대해서 디자인을 하였다. 첫 번째 디스크 할당 방식인 Linked Allocation 에서는 따로 빈 공간을 관리할 영역이 없어도 된다. 블록의 다음 링크를 찾을 수 있게 해주는 블록 링크 테이블을 통해 해당 블록이 비었는지 할당되었는지 판단할 수 있기 때문이다. [그림3.6]에서도 보듯이 데이터 블록의 인덱스가 결국엔 블록 링크 테이블의 인덱스와 같으며 빈 공간인 영역은 0으로 나타내어진다.

두 번째와 세 번째 디스트 할당 방식인 Contiguous Allocation과 +append는 할당되지 않은 연속된 영역을 관리해 주어야 한다. 이를 위해 [그림3.9] 와 같이 블록 비트맵을 두어 블록 별로 할당 여부에 대해서 비트맵으로 표시해주었다. 하지만 파일을 생성 시마다 빈 연속된 공간에 대해서 검사를 해야 하는데 이러한 비트맵으로만 유지하게 되면 검사하

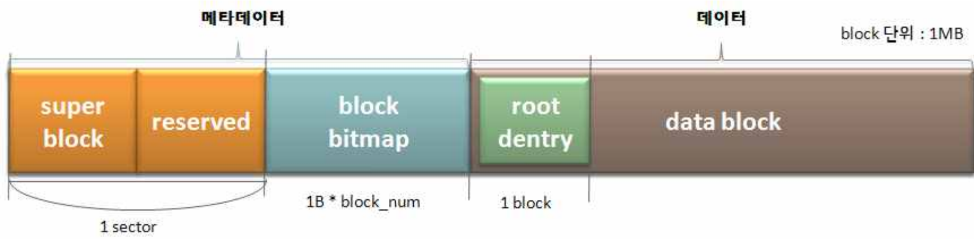

그림 3.9 Contiguous Allocation 파일시스템 전체구조

기 위해 디스크를 읽는 오버헤드가 커지게 된다. 그렇기 때문에 [3.10]와 같이 비트맵을 읽어 연속된 빈 공간을 리스트로 메모리에 저장하는

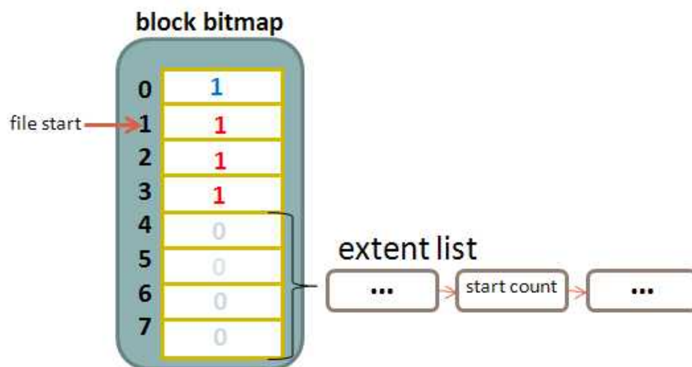

그림 3.10 Contiguous Allocation Block Bitmap 구조

방식을 디자인 하였다. Extent라는 연속된 빈 공간을 저장할 구조체를 디자인 하였고, 영역의 시작 인덱스와 블록 수를 저장할 수 있도록 하고 블록 수를 기준으로 오름차순 정렬을 통해 파일 생성 시 가장 영역이 덜 남는 연속된 영역을 선택할 수 있도록 하였다. 파일 삭제 시 Free Extent 리스트에 새로운 영역을 추가되며 앞이나 뒤쪽으로 연속된 영역일 경우에 merge 될 수 있도록 디자인하고 구현하였다.

제 4 장 Implementation Details

우리는 여러 스레드 환경에서 동시 쓰기 작업을 할 때의 성능 저하를 없애기 위해 본 연구를 진행하고 있으며 이전 섹션에서 설명한 디자인을 적용한 프로토타입 형태의 유저 레벨 파일시스템을 구현하였으며, 이를 어떻게 구현하였는지에 대해서 설명할 것이다. Low-level Function 섹션에서는 실제 파일시스템 구동을 위해 메타데이터나 데이터를 직접 하드디스크에서 조작하기 위한 함수들에 대해 설명할 것이며, High-level Function 에서는 Low-level Function들을 활용하여 사용자나 어플리케이션이 활용할 수 있는 인터페이스를 어떻게 구현했는지에 대해 말할 것이다. 또한 Implement Issue 섹션에서 구현 시에 방식이나 문제가 되었던 상황 등을 포함한다.

4.1 Low-level Function

앞에서 설명했던 바와 같이 우리는 기존 파일시스템을 거치지 않고 블록 사이즈를 크게 내려주기 위해 파일시스템 기능을 디자인 하였으며 이를 위해 파일시스템 전반을 조작하는 함수들을 구현하였다.

파일 시스템 초기화 함수는 사용하게 되는 하드 디스크(디바이스) 초기화하고 디바이스에 대한 fd 포인터 얻어온다. 이를 통해 하드디스크의 크기에 맞춰 각 메타 데이터와 데이터 영역의 디스크 시작 오프셋 및 크기 등을 계산하고 디스크의 Super Block 영역에 포맷한다. 또한 계산된 정보를 이용해 reserved 영역이 없다면 바로 block link table 영역(block개수*4B 0x00으로 초기화) 과 루트 디렉터리도 모두 0으로 초기화

한다. 그리고 파일시스템 인식 함수를 통해 파일 시스템 연결을 한다. 연결을 위해 디스크에 적혀진 파일 시스템 정보를 super block 구조체에 읽어온다.

블록 링크 테이블은 블록 링크 테이블 읽기 함수와 쓰기 함수 그리고 빈 블록 검색 함수, 링크 정보 반환 및 설정 함수로 제어한다. 블록 링크 테이블 읽기 함수와 쓰기 함수는 파일 시스템 자료구조(super block)에 저장된 블록 링크 테이블의 시작 어드레스를 더해서 블록 링크 테이블 내의 오프셋에서 섹터를 읽고 쓴다. 비트맵 제어 함수의 경우에도 이와 비슷한 형식으로 작성하였다.

빈 블록 검색 함수는 빈 블록으로 표시된 링크 정보 찾으면 블록 인덱스 반환한다. 링크 정보 반환 함수와 설정 함수는 블록 링크 테이블에서 해당 블록의 링크 정보를 반환하거나 설정하며 이를 위해 섹터단위로 오프셋을 사용하였다. 한 섹터에 128개의 블록 링크(4B)가 들어가므로 섹터 오프셋을 128로 나누면 섹터 오프셋을 구할 수 있다.

루트 디렉터리는 블록 읽기와 쓰기 함수와 빈 디렉터리 엔트리 검색 함수, 빈 디렉터리 엔트리 반환, 디렉터리 엔트리 설정 및 반환 함수, 디렉터리 엔트리 검색 함수 등으로 제어한다. 블록 읽기와 쓰기 함수는 데이터 영역의 시작 어드레스를 더해서 데이터 영역의 오프셋에서 한 블록(1MB)을 읽고 쓰며 빈 디렉터리 엔트리 검색 함수는 빈 디렉터리 엔트리 반환한다. 디렉터리 엔트리 설정(set) 함수 와 반환(get) 함수는 루트 디렉터리의 해당 인덱스에 디렉터리 엔트리 설정 및 반환하며 디렉터리 엔트리 검색 함수는 루트 디렉터리에서 파일 이름이 일치하는 엔트리를 찾아서 인덱스 반환한다. 예를 들어 Low-Level Function들을 이용해 파일이름을 넘겨받아 빈 파일을 생성하기 위해선 블록을 할당한 후 디렉터

리 엔트리를 할당하고 데렉터리 엔트리를 등록하면 된다. 반대로 삭제하기 위해선 디렉터리 엔트리를 검색하고 블록을 반환하여 디렉터리 엔트리를 삭제하면 된다.

4.2 High-level Function

유저레벨에서 파일을 조작할 시에 파일을 먼저 열고 사용하게 된다. 파일을 열 때는 파일을 생성하거나 비우는 역할도 겸한다. 파일을 열 때 수행하는 작업은 mode 파라미터에 의해 결정되며 mode에 따라 읽기나 읽기와 쓰기로 파일을 열고, 파일 생성 등의 작업을 함께 수행한다. 다음

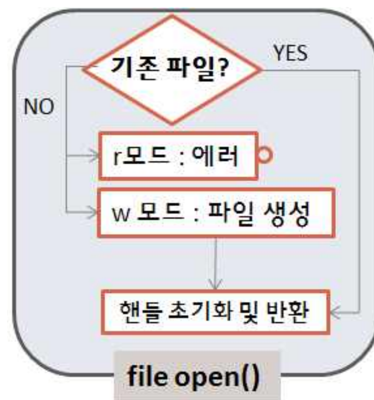

그림 4.1 파일 열기 도식

[그림4.1] 이 파일 열기 함수를 간단한 도식표로 나타낸 것이다. 루트 디렉터리에 해당하는 파일의 디렉터리 엔트리가 존재하는지 확인하여 파일을 생성하거나 비우는 일을 하게 된다. 이때 루트 디렉터리에 파일이 존재하지 않으면 읽기 작업은 불가능하므로 프로그램을 종료시키고 쓰기 작업의 경우 파일을 생성하고 파일의 디엔트리 에서 핸들에 필요한 정보를 저장하고 핸들을 반환하게 된다.

파일을 쓸 때는 레코드의 크기와 개수를 곱한 크기만큼을 버퍼에서 파일로 쓴다. 파일 쓰기 함수는 파일을 확장하는 부분과 파일 크기를 업데이트를 하는 부분을 제외하면 파일 읽기 함수와 비슷하다. 다른 점이라면 쓰기를 수행하기 전에 쓸 수 있는 블록이 있는지 확인한 후에 처리 여부를 검사한다는 점이다. 또한 파일 오프셋이 블록의 시작 위치에 있는지 확인한다. 써야할 바이트 수와 파일 오프셋의 시작 오프셋을 검사하는 이유는 블록 단위로 작업을 하기 때문이며 데이터를 쓰는 위치가 블록의 시작이 아니거나 클러스터 하나를 모두 채우지 못하면 블록의 일부만 변경되므로 디스크에서 블록의 읽은 후에 수정된 부분만을 채워야 하는 단점이 있다. 만약 써야 할 데이터가 많다면 블록 시작 오프셋의 배수에 맞춰 쓰기 함수를 호출 하는 것이 좋다. 만약 클러스터를 더 할당해야할 경우가 생긴다면 Linked Allocation 에서는 Blocklink Table을 검사하여 빈 블록을 찾아 할당하며, CA+append 같은 경우에도 Free Extent List를 보고 빈 연속된 블록을 찾아 할당하여 dentry를 추가하여

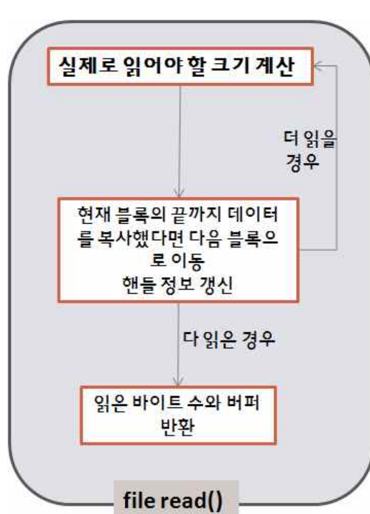

그림 4.2 파일 읽기 도식

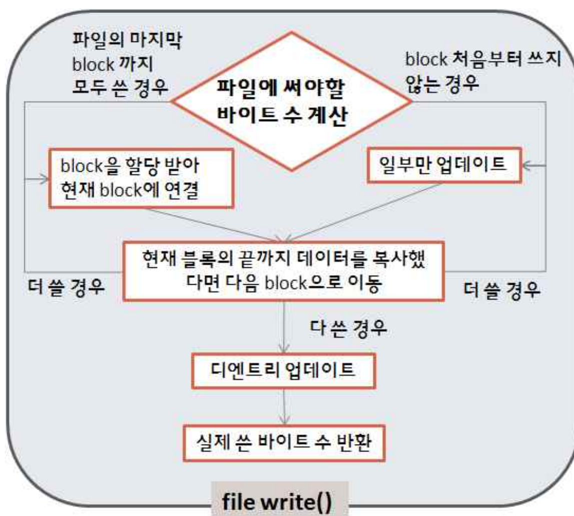

그림 4.3 파일 쓰기 도식

준다. [그림4.2]이 파일 쓰기 함수의 기본 처리과정을 도식화한 것이다.

읽기 함수를 진행할 시에는 실제로 읽어야할 크기를 먼저 계산하고 파일의 시작부터 끝 방향으로 포인터를 이동시키면서 읽게 된다. 만약에 해당 블록의 데이터를 끝까지 복사했다면 핸들 정보를 갱신하고 더 읽을 바이트가 남아있을 때 다음 블록을 읽게 된다. 다 읽게 되면 읽은 바이트 수와 버퍼를 반환한다. [그림4.3] 이 파일 읽기 처리의 도식도이다.

4.3.1 Implementation Issue

Sequential Write 할 시에 성능 저하 문제 (50~60MB/sec)가 있었는데 상태를 확인한 결과 write할 시 read bandwidth가 존재함을 발견할 수 있었다. 커널에서 disk를 읽을 때 시작 오프셋을 4096 단위로 맞춰주지 않으면 영역을 읽은 후 쓰게 되기 때문에 일어나는 현상임을 확인하였고, 시작 offset을 맞추며 쓰는 단위도 4096의 배수로 오프셋을 align시켜주어 해결해 주었다.

제 5 장 실험 및 검증

우리는 한 개의 머신을 사용하여 실험을 진행하였다. 하나의 Intel(R) Core(TM) i5 760 2.80GHz quad core CPU와 main memory 8G를 가지고 OS용 1000.2 GB와 실험용 디스크 2000.4 GB를 장착하고 Ubuntu 12.04(Linux kernel 3.2.37) 환경에서 진행하였다.

우리는 기존 파일시스템(ext2)과의 성능을 비교했고 제안한 디자인 중에서 Contiguous Allocation을 기반으로 실험을 진행하였다.

5.1 Sequential Write Test

우리는 앞에서 설명했던 바와 같이 블록의 사이즈를 늘려 멀티 스레드 간의 간섭으로 인한 순차적 디스크 접근을 장려하였으며 본 연구에서 구현한 파일 시스템을 통해 기능 테스트를 해보았다. [표5.1]에서는 스레드를 늘려가며 sequnetial한 write 실험을 진행한 결과이다. 요청은 64G로 하였고 사용한 버퍼 사이즈는 1MB로 동일하게 사용하였다. 결과를

		thread1	thread2	thread3	thread4
my_fs	Time(sec)	460.9	461.4	461.9	464.0
	Throughput(MB/s)	145.6	145.4	145.2	144.6
ext2	Time(sec)	474.4	587.9	632.8	689.3
	Throughput(MB/s)	141.4	114.1	106.0	97.3

표 5.1 sequential write test time & throughput

보면 우리가 구현한 파일 시스템의 경우 스레드가 늘어남에 따라 시간이나 성능에 차이가 거의 없음을 확인 할 수 있었고, ext2 같은 경우에는 스레드 수가 늘어남에 따라 성능의 저하를 보였다. [그림5.1]의 그래프를

보면 차이를 더 확실히 알 수 있으며 구현한 프로토타입 파일시스템이 ext2 보다 스레드가 4개 일 때 48%정도의 성능 향상을 보임을 확인 할 수 있었다. 즉, 블록 사이즈가 늘어남에 따라 디스크의 효율성이 향상됨을 확인할 수 있었고 해당 파일시스템이 그 효과를 보여주었다.

5.2 Benchmark Test

우리는 FIO와 IOZONE과 같은 실제 micro-benchmark tool을 이용하여 [7], 실제 Sequential Write과 read pattern에 대한 실험을 하였다. 기본적으로 기존의 리눅스 파일시스템인 ext2에서 수행한 실험결과와 본 연구에서 제안한 프로토타입 파일시스템을 통해 수행한 실험결과를 비교하였다.

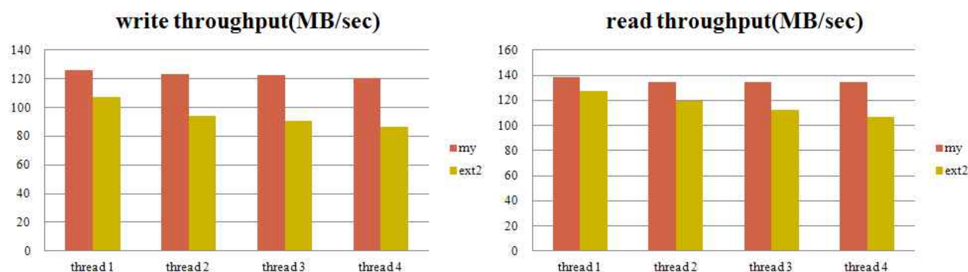

그림 5.1(a) fio write　　　　그림 5.1(b) fio read

[그림5.2(a)]와 [그림5.2(b)]를 보면 write, read 모두의 결과 값이 우리가 작성한 synthetic benchmark 와 유사하게 나옴을 확인 할 수 있다. ext2가 더욱 쓰레드 수에 영향을 받으며 구현한 프로토타입 시스템이 기존보다 성능의 저하가 줄어듬을 확인하였다. IOZONE을 통해 실험한 결과는 아래 [그림5.3(a)] [그림 5.3(b)]와 같다.

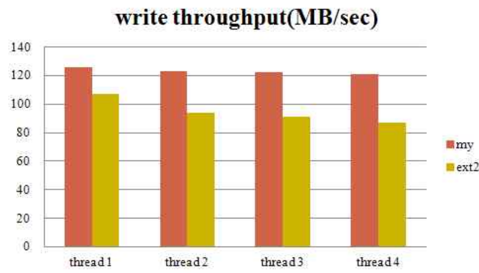

그림 5.2(a) iozone write

그림 5.2(b) iozone read

IOZONE 벤치마크에서도 유사한 경향을 나타냄을 확인하였다. write의 경우 쓰레드 수가 많을 경우 ext2 보다 프로토타입이 21% 성능 향상을 보였다.

제 6 장 결론

우리는 멀티 스레드 환경에서 sequential write에 대한 디스크의 효율을 살릴 수 있는 파일 시스템에 대한 디자인 및 구현하였다. 데이터 블록이 작을 때 성능의 저하가 이으므로 이를 크게 해주고 큰 블록을 다루는 파일 시스템의 기능을 구현하기 위해 디스크 할당, 빈 공간 관리와 같은 디자인을 하였으며 디렉터리 구조와 경로 처리의 복잡함을 줄이기 위해 트리 형태의 계층적은 구조 대신 수평적인 구조로 설계하였다. 파일 시스템 초기화와 블록 링크 테이블 제어 및 비트맵 제어와 루트 디렉터리 제어 등의 Low-level Function을 통해 파일 시스템의 밑단을 조작할 수 있도록 하였고, 파일 열기, 파일 쓰기, 파일 읽기, 파일 삭제 등의 High-level Function을 통해 우리가 구현한 프로토타입 파일시스템을 유저가 사용할 수 있도록 하였다.

우리는 위에서 언급한 디자인들을 프로토타입 형태로 구현해 기존 파일 시스템 ext2와의 비교 실험을 진행하였으며, 스레드 수가 많아질 때 sequential write에 대해서 약 48%의 작업률 향상과 여러 스레드가 디스크 공유함에 있어서 발생하는 성능 저하가 줄어듬을 볼 수 있었다.

참고 문헌

[1] “Apache Hadoop”, http://hadoop.apache.org/

[2] J. Dean and S. Ghemawat, “MapReduce: Simplified Data Processing on Large Clusters”, USENIX Association OSDI ’2004.

[3] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," Mass storage Systems and Technologies, IEEE / NASA Goddard Conference on, vol. 0, pp. 1-10, 2010.

[4] Card, R., Ts’o, T., Tweedie, S.: Design and Implementation of the Second Extended Filesystem. In: First Dutch International Symposium on Linux 1994

[5] J. Shafer, S. Rixner, and A. L. Cox. "The Hadoop Distributed Filesystem: Balancing Portability and Performance", In Proceedings of the 2010 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS’10), pages 122 – 133, 2010.

[6] Y. Xu, G. Sun, S. Jiang, "intelliQos: Rethinking Storage QoS Implementation for System Efficiency", EuroSYS’ 2013.

[7] http://www.bluestop.org/fio/HOWTO.txt

ABSTRACT

Multi-thread on multi-core platform recently has been come into wide use. However, it is often reported that sequential writing of the multiple thread cannot provide enough bandwidth because random seek time increases. File system cannot maintain concurrent sequential write because of small block size. Thus, it occurs less bandwidth with multiple thread.

This paper proposes a prototype file system supporting large block. priority boosting and Load balancing for improving interactivity of android smartphones. The proposed techniques organize file system with larger(1MB) block than original block.

The experiment results demonstrate prototype file system improve sequntial write performance up to 48% compared to the legacy system.

Keywords : File System, Large Sequantial Write, Disk Allocation, Free Space Management, Multithread

Student Number : 2011-20945

감사의 글

지난 연구실 생활을 돌이켜 보니 아쉬움이 많이 남습니다. 여러가지로 부족한 제가 무사히 연구실 생활을 마칠 수 있었던 것은 주위 많은 분들의 도움이 있었기 때문이라고 생각합니다. 힘들어서 포기하고 싶고 나태해지는 순간도 분명 존재했지만 마음을 다잡고 끈기를 잃지 않았던 건 곁에 든든히 있어주신 선후배님들 덕분입니다. 장난스러움이 담긴 따스한 말들로 저를 여기까지 이끌어 주신 점 감사합니다. 학문적인 면에 있어서 제가 남들에게 자랑스럽게 내밀만한 무언가를 이룬 것이 없어서 아쉽지만 내적인 면에서는 스스로 성장했음을 느낍니다. 지금은 비록 새로운 환경에서 또 다시 신입이라는 간판을 걸고 시작하는 입장으로서 연구실에서의 경험을 바탕으로 점점 더 나아지는 모습 보여드릴 수 있기를 기원합니다.

졸업하는 이 순간까지 저와 함께 해주신 많은 분들께 감사드립니다. 특히 늘 옆에서 걱정해주시면서도 저를 믿어주신 부모님께 감사드립니다. 그리고 부족한 저를 이끌어 주시고 연구에 정진할 수 있도록 해주신 두 분, 염헌영 교수님과 엄현상 교수님께 감사드립니다. 항상 진심으로 챙겨주시고 유쾌했던 신규오빠, 곁에 있는 것만으로도 든든했던 내영언니, 능력자이신 신웅오빠, 누가 뭐래도 한국인 같은 계신이, 성실의 아이콘 동유오빠, 편의점 음식은 이제 조심! 민영언니, 누가 뭐래도 경비대장 성재오빠, 정직하고 솔직해서 빵빵 터졌던 문봉이, 한결같이 엘레강스하셨던 설웅오빠, 수다친구 용석오빠, 날 괴롭히면서 삶의 낙을 찾았던 명원오빠, 배려심 깊은 지웅오빠, 이쁘고 재밌는 한울이, 나 없어도 우울해하지 말고 화이팅 다혜, 이름 말투 모두 특이한 시봉오빠, 이쁜 아내분과 행복하실 성구오빠, 또 재밌게 놀았던 동갑내기 둘기 친구 찬호, 무뚝뚝해보이지만 은근 수다쟁이 재우오빠, 묵묵히 할 일을 했던 국태오빠, 연

륜이 느껴지셨던 세훈오빠, 모두 함께 지내고 성장할 수 있어서 즐거웠습니다.

지금은 함께 있지 않지만 항상 웃음을 잃지 않았던 유느님 영진오빠, 회사 바삐 다니면서도 잘 챙겨주시고 도움주신 은성오빠, 연구실의 든든한 맏형으로 있어주셨던 동인오빠, 새침떼기같지만 정이 많았던 형석오빠, 일과 영어 모두 잘했던 용경오빠, 브랜드에 빠삭한 패셔니스타 완희오빠, 모르는 게 없었던 구글 검색엔진 승미언니, 친절하고 여성스러운 승민언니, 깨끗하고 섬세한 맏언니 인순언니, 학기 중에도 바쁘셨던 민규오빠, 모두 그립고 감사했습니다.