工學碩士 學位論文

指導敎授 李 珉 秔

개인용 멀티미디어 시스템을 위한 리눅스 파일 시스템 구현

The Implementation of Linux File System for Personal Multimedia System

2003年 12月

漢城大學校 大學院

컴퓨터 工學科

컴퓨터 工學 專攻

孫 正 洙

工學碩士 學位論文

指導敎授 李 珉 秔

개인용 멀티미디어 시스템을 위한 리눅스 파일 시스템 구현

The Implementation of Linux File System for Personal Multimedia System

위 論文을 工學碩士 學位論文으로 提出함

2003年 12月 日

漢城大學校 大學院

컴퓨터 工學科

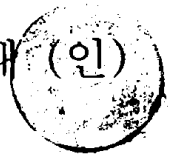
컴퓨터 工學 專攻

孫 正 洙

손정수의 공학석사 학위논문을 인정함

2003年 12月 日

심사위원장 김 영 웅 

심사위원 황 기 태 (인) 

심사위원 정 인 환 

목 차

표 목 차

그 림 목 차

부 록 목 차

I. 서 론

오늘날 IT 발전이 PC가 아닌 내장형 시스템에 집중 되면서 DVR, PVR, PDA, 셋톱박스 등 주로 대형 멀티미디어 파일을 저장하고 재생하는 개인용 멀티미디어 시스템이 급속히 발전하고 있다. 또, 이러한 장치들이 주로 리눅스를 운영체제로 사용하고 있다. 그러나 기존 리눅스 파일 시스템들은 상대적으로 작은 파일의 입출력에 최적화 되어 있어 잦은 전원 차단과 대형 멀티미디어 파일을 다루는 개인용 멀티미디어 시스템에는 성능과 안전성면에서 적합하지 않다.

본 논문에서는 멀티미디어 파일을 저장하고 재생하며 높은 안정성을 보장하기 위한 새로운 멀티미디어 파일 시스템인 PMFS (Personal Multimedia File System)를 구현하였으며, 다양한 실험으로 성능 평가를 실시하여 PMFS의 성능을 검증하였다. 개발된 파일 시스템은 약간의 수정으로 리눅스가 아닌 다른 운영체제에도 적용 가능하다.

본 논문의 구성은 다음과 같다. 2장에서는 관련 연구로 기존 리눅스 파일 시스템의 종류와 대형 멀티미디어 파일이 저장되는 개인용 시스템에서 기존 파일 시스템들의 문제점을 언급하며, 3장에서는 PMFS의 구조 및 구현 내용을 기술한다. 4장에서는 PMFS 대한 성능 평가를 실시하며 5장에서는 결과 분석을, 6장에선 본 논문의 결론을 기술함으로 본 논문을 마치고자 한다.

II. 관련 연구

1. 리눅스 파일 시스템

파일 시스템은 컴퓨터의 하드 디스크에 데이터를 저장하고, 회수하는 운영체제를 위한 논리적인 수단이다. 즉, 파일의 자료구조, 데이터가 디스크에 저장되는 방식, 파일을 읽고, 쓰는 연산 등 파일에 관련된 모든 것을 관리한다[Bar, 2001][권수호, 2002].

리눅스 파일 시스템은 1991년 리눅스가 탄생한 이래 지난 수년 동안 지속적인 발전을 거듭해 왔다. 그 중 오늘날 가장 널리 사용되는 파일 시스템은 리눅스의 표준 파일 시스템인 EXT2와 저널링 파일 시스템들인 EXT3, JFS, XFS, ReiserFS이 있다.

1.1 EXT2

EXT2 (The Second Extended File System)는 Way Davidson에 의해 설계된 리눅스 표준 파일 시스템으로 일반 파일, 디렉토리, 디바이스 파일, 심볼릭 링크의 표준 Unix 파일 타입을 지원한다. 또한 4TB까지의 대용량의 파티션 지원이 가능하며 255 문자까지의 긴 파일 이름을 지정할 수 있고 필요에 따라 1012 문자까지 확장이 가능하다[Bar, 2001][Card외, 1994].

1.2 EXT3

EXT3 (The Third Extended File System)파일 시스템은 Stephen Tweedie가 설계한 것으로 EXT2에 저널링을 도입한 EXT2의 확장 파일 시스템이다[Tweedie, 2000]. EXT2와 똑같은 디스크 포맷과 메타 데이터를 가지고 있으며 메타 데이터 및 데이터 저널링을 지원하고 순차적인 파일 이름 찾기가 가능한 블록 기반이다[Robbins][Bryant외, 2002].

1.3 ReiserFs

ReiserFS는 Namesys의 Hans Reiser와 그가 이끄는 팀이 개발한 파일 시스템으로 작은 파일 엑세스 성능 향상에 초점을 맞추어 설계되었다. ReiserFS는 메타 데이터 저널링하며 디렉토리, 파일, 데이터를 구성하는 방법으로 B* 트리를 사용한다[Bryant외, 2002].

1.4 JFS

JFS (Journaled File System)는 IBM 사에 의해 제작된 파일 시스템으로 현재 IBM의 엔터프라이즈 서버에서 사용되고 있다. JFS는 서버 환경에서의 높은 처리량을 위해 설계된 파일 시스템으로서 메타 데이터 저널링을 지원하며 메타 데이터를 B+ 트리로 관리한다. 또한, 동적 아이노드 할당 방법을 사용하여 사용되지 않는 아이노드는 해제하여 디스크 저장 공간을 절약할 수 있다[Bryant외, 2002][권우일외][Journaled].

1.5 XFS

XFS는 SGI 사에 의해 개발되었으며 IRIX 운영체제에 사용되고 있다[XFS]. XFS는 블록 크기를 512B에서 64KB까지 늘릴 수 있고, 이를 통해 대용량 파일을 저장하는 서버를 위해 설계되었으며 메타 데이터 저널링을 지원한다. 메타 데이터 관리를 위해 B* 트리를 사용하며 완전한 64-bit 파일 시스템으로 수백만 테라 바이트 용량을 지원한다[Bar, 2001][Bryant외, 2002][Sweeney, 1996].

2. 개인용 멀티미디어 시스템을 위한 파일 시스템

2.1 멀티미디어 파일의 특징

멀티미디어 파일이라 함은 숫자나 문자 위주의 데이터를 벗어나 텍스트, 그래픽, 이미지, 오디오, 비디오, 애니메이션 등 여러 미디어 데이터 정보가 디지털 방식으로 융합된 형태의 파일을 말한다[멀티]. 멀티미디어 파일은 다음과 같은 특징을 갖는다.

1) 대형이다. 멀티미디어 파일은 주로 음악, 영상 등의 정보를 저장하므로 한 파일이 수십 메가 바이트에서 수기가 바이트까지 매우 큰 데이터로 유지된다.
2) 파일에 대한 읽기, 쓰기 연산이 주로 순차적으로 일어난다. 멀티미디어 파일은 여러 가지 데이터 형태가 복잡하게 얽혀 있다. 하지만 이 파일에 대한 연산은 빨리 감기, 재생, 빠른 재생, 저장 등 각각의 미디어 데이터에 대해 순차적 읽기, 쓰기 형태로 이루어진다.
3) 읽기 연산 중심이다. 대표적인 멀티미디어 파일인 영화, 애니메이션, 음악 파일들은 한번 디스크에 기록 된 후, 계속해서 읽혀 재생되는 읽기 연산이 대부분을 차지한다.
4) 멀티미디어 데이터는 특별한 형식이 없는 비정형의 구조를 갖는다. 이미지 데이터의 경우 일정한 구조를 갖지 않는 일정 비트의 연속적인 스트림(stream)으로 볼 수 있다.
5) 시간성을 갖는다. 오디오와 비디오, 애니메이션 데이터는 본질적으로 시간에 의존하는 특성을 가지며, 여러 가지 멀티미디어 데이터가 복합적으로 시간적인 순서에 따라 표현되는 동기화 특성이 존재한다.

이중 파일 시스템의 성능과 관련이 있는 것은 1),2),3) 이다. 나머지 것들은 응용프로그램에서 멀티미디어 파일에 대한 동기화와 관리 기법과 연계가 있다. 따라서 본 연구에서는 멀티미디어 파일 가운데 파일 시스템의 성능과 관련이 있는 1),2),3)의 특성에 대해 고려하며 분석하고 이를 지원하

기 위한 기존 파일 시스템의 문제점을 제시하고자 한다.

2.2 개인용 멀티미디어 시스템에 대한 기존 파일 시스템의 문제점 (EXT2, EXT3)

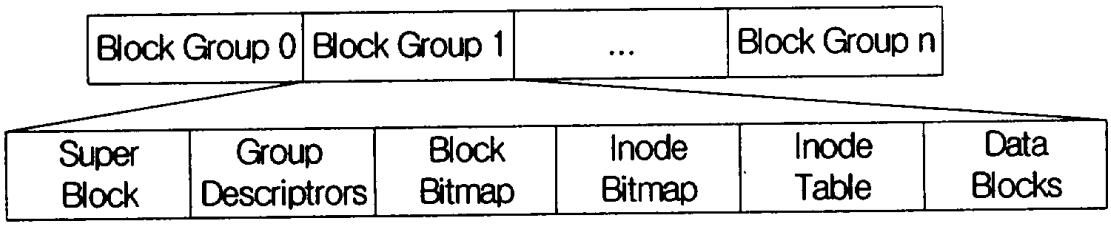
현재 리눅스에서 가장 많이 사용되고 있는 파일 시스템으로는 EXT2와 EXT3이 있으며 이들이 대형 멀티미디어 파일을 지원하는데 가지는 문제점은 다음 세 가지로 요약할 수 있다.

1) 디스크 상의 데이터 배치로 인한 문제
2) 데이터의 트리 구조 관리로 인한 문제
3) 버퍼 캐쉬 사용으로 인한 문제

멀티미디어 데이터를 효과적으로 읽고 쓰기 위해서는 디스크 헤드의 이동 시간(seek time)을 최소화하는 것이 가장 중요하다. 디스크 헤드의 이동 시간은 데이터 블록의 배치 방식과 메타 데이터의 구조에 따라 결정되며 이는 파일 시스템에 의해 정의된다[원유집외]. 따라서 멀티미디어 시스템을 위한 파일 시스템은 멀티미디어 파일의 특성을 감안한 데이터 블록 배치 방식과 이를 반영한 메타 데이터의 구조를 갖도록 설계되어야 한다. 그러나 유닉스 파일 시스템에 많은 영향을 받은 EXT2는 텍스트 기반의 비교적 작은 파일의 처리 성능 향상을 중심으로 발전하였다. 이로 인해 개인용 멀티미디어 시스템에 적용하는데 심각한 성능 상의 문제점이 발생하게 된다.

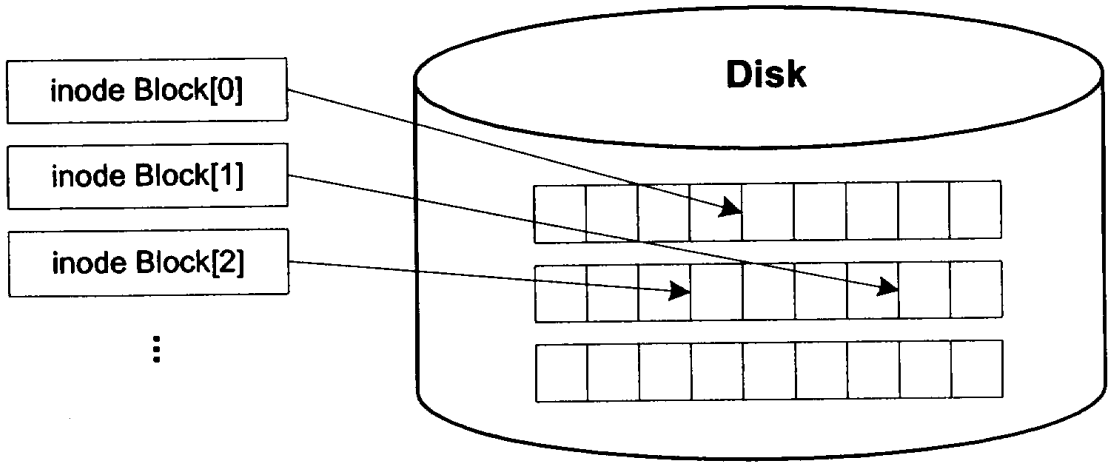
2.2.1 디스크 상의 데이터 배치 방법에 의한 문제

EXT2는 (그림 1)과 같은 디스크에 데이터 블록을 배치한다.



(그림 1) EXT2의 디스크 데이터 배치

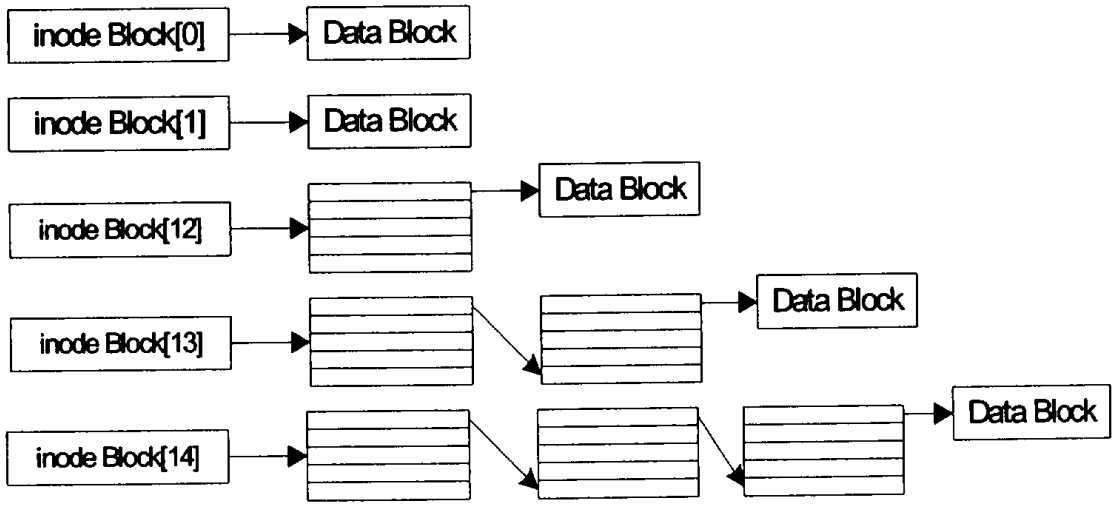
EXT2는 디스크 헤드의 이동 시간을 줄이기 위해 블록 그룹 메커니즘을 사용하여 데이터 블록들을 관리한다. 블록 그룹은 (그림 2)와 같이 일련의 연속된 실린더들의 그룹으로 데이터 블록을 상대적으로 좀더 가까운 실린더 위치에 배치하여 디스크 탐색의 효율성을 향상시키는 메커니즘이다. 이는 사이즈가 작은 여러 파일들을 관리할 경우 시스템의 성능을 효과적으로 증가시킨다. 그러나 파일의 대부분이 블록 그룹의 사이즈보다 훨씬 크고 높은 순차적 접근 특성을 가진 멀티미디어 파일의 경우에도, 파일은 여러 개의 다른 블록 그룹으로 나누어져 저장되며 데이터 블록은 디스크 상에 분산되게 된다. 이로 인해 디스크 헤드는 좀더 빈번히 움직이게 되어 디스크 탐색 오버헤드가 발생한다[원유집외].



(그림 2) EXT2의 블록 기반 할당 구조

2.2.2 데이터의 트리 구조 관리로 인한 문제

EXT2의 블록 그룹은 슈퍼 블록, 그룹 디스크립터, 비트맵, 아이노드 테이블, 데이터 블록으로 구성되어 있다. 슈퍼 블록은 EXT2 파일 시스템을 유지하고 운용하는데 이용되는 매우 중요한 정보이며, 그룹 디스크립터는 전체 블록 그룹들의 상태를, 아이노드 비트맵은 해당 블록 그룹 내에 존재하는 아이노드의 사용 유무를, 아이노드 테이블은 블록 그룹 내 여러 아이노드를 지정하며 데이터 블록은 실제 데이터에 대한 논리적 모델을 나타낸다. 이중 데이터의 관리 구조를 결정하는 것이 아이노드이다. 아이노드는 EXT2 파일 시스템에서 데이터 블록들에 대한 포인터 및 관련 정보를 가지는 것으로 EXT2는 이 아이노드를 통해 (그림 3)과 같은 트리 구조로 파일 시스템 내 모든 객체를 관리한다. 이러한 트리 구조는 작은 데이터 블록이 여러 군데 흩어져 있을 경우 데이터 탐색에 있어서 매우 효과적일 수 있다. 그러나 순차적 읽기 쓰기를 수행하는 멀티미디어 파일의 경우 불필요한 디스크 검색 오버헤드를 발생시켜 시스템의 성능 저하를 가져오는 원인이 된다. 또한 데이터 블록을 관리하기 위해 아이노드와 같은 메타 데이터를 많이 유지할 경우 그 만큼 많은 디스크 I/O를 필요로 하게 되어 시스템의 성능을 떨어드린다[Bar, 2001].



(그림 3) EXT2의 트리 구조

2.2.3 버퍼 캐쉬 사용으로 인한 문제

멀티미디어 데이터를 위한 EXT2의 또 다른 문제는 버퍼 캐쉬의 사용으로 인해 발생한다. 버퍼 캐쉬는 데이터의 재사용 가능성을 고려하여 메모리의 일정 부분을 캐쉬로 사용하는 정책이다. 이는 서버와 같이 많은 프로그램이 같은 파일을 재사용할 때 시스템의 성능을 크게 향상시켰다. 그러나 단일 프로그램이 순차적으로 데이터에 엑세스하는 멀티미디어 시스템의 경우 오히려 메모리 낭비와 많은 스왑핑으로 인한 시스템의 성능 저하를 가져오는 원인이 된다. 또한, 서버와 달리 불시에 전원을 끄는 사례가 빈번하게 발생하는 가전 형태의 멀티미디어 시스템의 경우 미처 디스크에 기록되지 못한 많은 데이터가 한 순간에 깨져 데이터의 무결성을 보장하지 못하는 원인이 된다. 심각한 경우 파일 시스템 전체가 깨질 수 있고 복구에도 오랜 시간이 걸리게 된다.

로그 파일 시스템[Czezatke, 2000] 기반인 ReiserFS, XFS, JFS와 같은 저널링 파일 시스템을 사용하면 복구 속도와 안정성, 그리고 XFS의 경우 순차적 연산 능력은 어느 정도 올라가지만 저널링을 위한 메타 데이터의 주기적 기록 때문에 시스템 성능이 떨어진다. 실제로 권우일 등이 행한 연구에서 DVR과 같은 멀티미디어 환경을 대상으로 시험한 결과, 저널링 파일 시스템들은 EXT2보다 전반적으로 낮은 읽기 성능을 보였다[권우일외].

위와 같은 문제는 기존 리눅스 파일 시스템이 대용량, 순차적 쓰기/읽기, 읽기 연산 중심이라는 멀티미디어 파일의 특성과 제한된 응용 프로그램 및 불시 전원 차단이라는 개인용 시스템의 특징을 충분히 고려하지 못했기 때문이며 이는 기존 리눅스 파일 시스템이 대형 멀티미디어 파일을 주로 사용하는 개인용 멀티미디어 시스템에 적합하지 않음을 보여준다.

이 논문에서는 개인용 멀티미디어 시스템을 위한 새로운 파일 시스템을 설계, 구현하였다.

III. 개인용 멀티미디어 시스템을 위한 리눅스 파일 시스템, PMFS 구현

1. 개인용 멀티미디어 파일 시스템의 요구 사항

멀티미디어 시스템을 위한 파일 시스템이 갖추어야 할 두 가지 중요한 요구 사항은 아래와 같다.

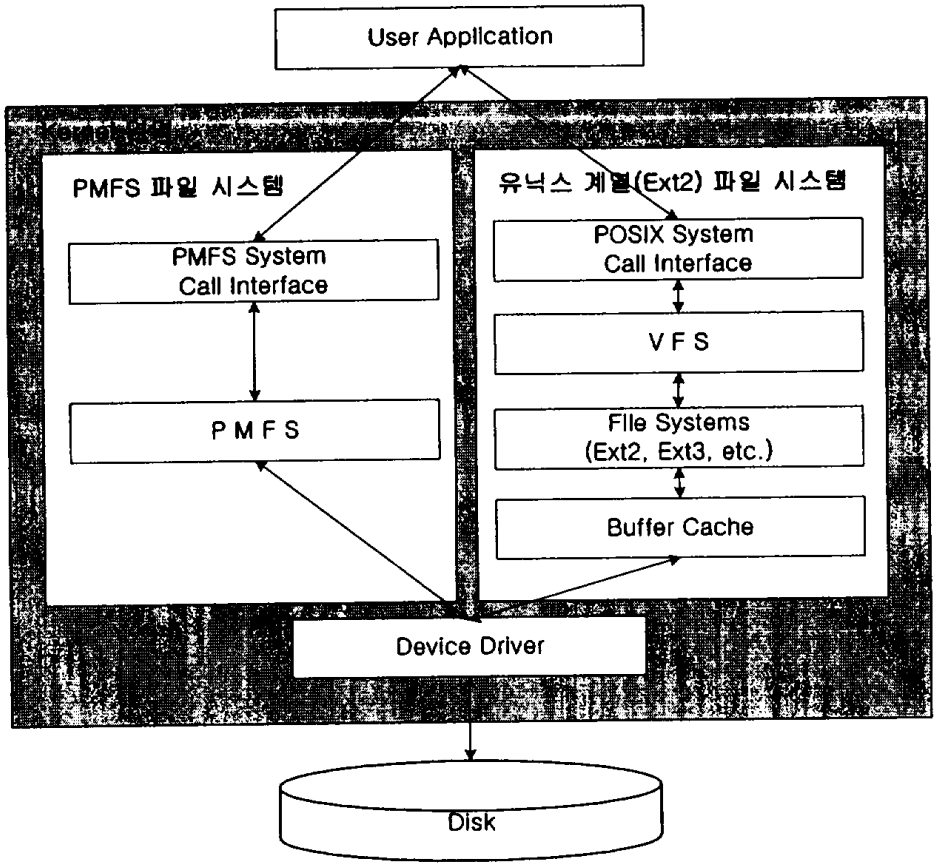
1) 대형 멀티미디어 파일을 위한 대용량의 블록 사이즈 구조
2) 순차적 데이터의 빠른 탐색을 위한 단순한 인덱스 구조
3) 빠른 읽기/쓰기 성능
4) 불시의 전원 차단에도 안전한 데이터 무결성

본 연구에서는 개인용 시스템에서 대형 멀티미디어 파일을 처리하기 위하여 기존 파일 시스템들의 단점을 보완하고 위 요구 사항을 만족하는 새로운 파일 시스템인 PMFS을 제안한다.

2. PMFS의 구조

2.1 PMFS의 전체 구조

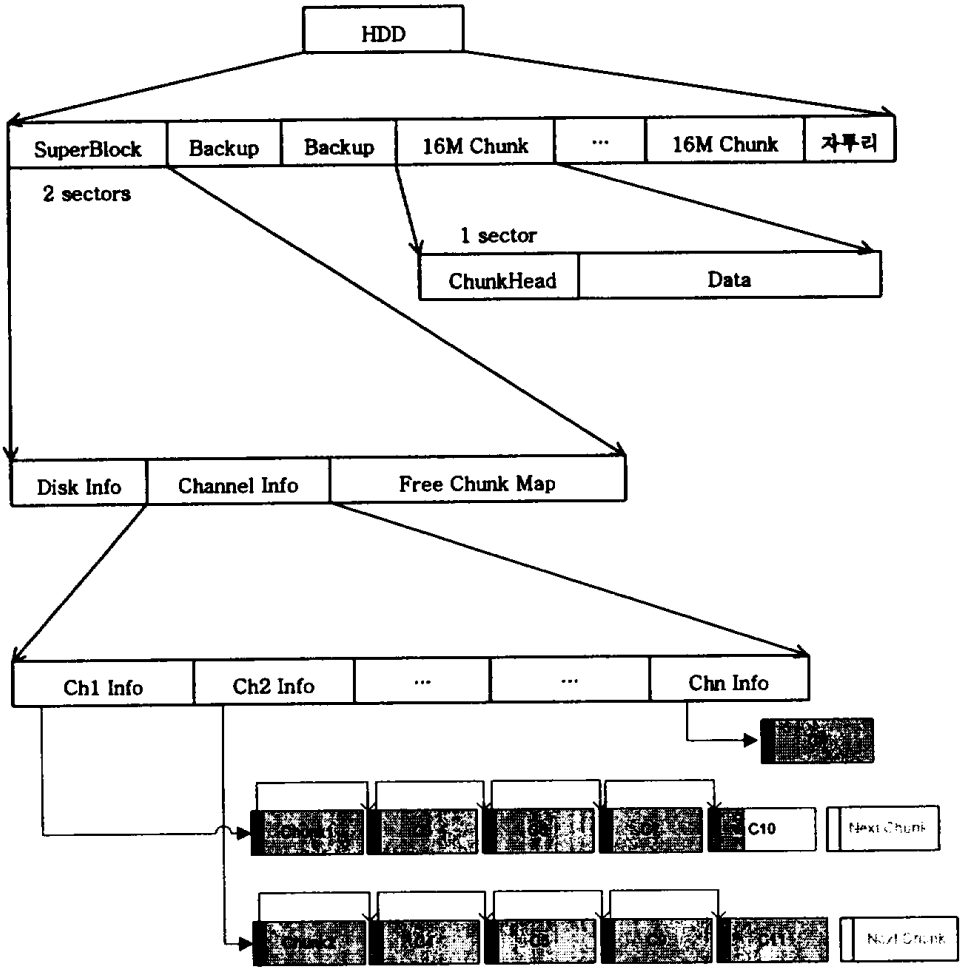
운영 체제에서 PMFS의 전체 구조는 (그림 4)와 같다. PMFS는 유닉스 계열 파일 시스템과 달리 자체 시스템 콜 인터페이스를 제공한다. 또 일반 유닉스 계열 파일 시스템은 디스크 블록에 대한 재사용을 대비한 성능 향상을 위해 버퍼 캐쉬를 파일 시스템과 디스크 사이에 두게 되며 데이터를 디스크에 직접 쓰지 않고 캐쉬를 통하여 관리한다. 하지만 이는 불시 전원 차단 시 데이터의 무결성을 보장하지 못하는 원인이 된다. 그러나 PMFS는 최소한의 독립된 버퍼를 유지하며 대부분의 읽기 쓰기를 버퍼 캐쉬 없이 디스크에 바로 수행한다.



(그림 4) PMFS의 전체 구조

2.2 디스크 상의 데이터 배치

PMFS는 순차적 쓰기/읽기 특징의 멀티미디어 특성을 반영하여 (그림 5)와 같은 데이터 배치 구조를 갖는다.



(그림 5) PMFS의 디스크 데이터 배치

PMFS는 하드 디스크를 하나의 슈퍼 블록과 이를 위한 백업 그리고 최소 4MB, 최대 16MB Chunk들로 구성한다. 슈퍼 블록은 파일 시스템 전체 상황과 정보를 유지하는 중요한 데이터로 디스크의 시작 섹터에 저장되며 만일의 사태에 대비하여 두 번 백업을 한다. 슈퍼 블록에 기록되는 데이터는 <표 1>과 같다.

<표 1> 슈퍼 블록 데이터 구조

FS_ID	파일 시스템 아이디
ChunkSize	Chunk의 사이즈
Chunk1Start	첫번째 Chunk위치
NumChunk	전체 Chunk 수
NumFreeChunk	빈 Chunk 수
ChannelInfo[채널수]	채널 정보
Checksum	슈퍼 블록 체크 섬
FCB[FCB_SIZE]	Free Chunk 비트맵

슈퍼블록이 수정될 때는 다음과 같은 세 경우일 때이다.

1) 새로운 Chunk가 할당 될 때
2) 빈 Chunk가 생겼을 때
3) 채널 정보 영역의 DataSize가 수정되는 경우 즉, 어느 채널이고 매번 데이터를 쓸 때 (이 때는 바로 쓰는 것이 아니고 적당한 delay 후에 기록한다.)

또, 슈퍼 블록에는 <표 2>와 같은 채널 정보를 채널 수만큼 유지한다.

<표 2> 채널 데이터 구조

MediaType	저장된 데이터 종류
ChunkCount	채널에 할당된 Chunk 수
StartTime	채널의 첫 데이터의 Time Stamp
FirstSeq	첫 번째 Chunk의 순서 번호
LastSeq	마지막 Chunk의 순서 번호
FirstChunk	첫 번째 Chunk의 번호
SecondChunk	두 번째 Chunk의 번호
LastChunk	마지막 Chunk의 번호
Last-1Chunk	끝에서 두 번째 Chunk의 번호
DataSize	채널의 전체 데이터 크기

채널은 PMFS가 데이터를 관리하는 단위로 대부분의 파일 시스템들이 파일을 유지하며 파일에 대해 읽기 쓰기를 수행하는 것과 같이 PMFS에는 데이터를 채널 단위로 관리하며 사용자는 파일을 사용할 때와 같이 해당 채널을 열고 채널에 쓰기/읽기를 수행하게 된다. 이 채널에 대한 데이터 구조는 슈퍼 블록 내에 존재하며 채널 수는 파일 시스템을 생성할 때 사용자가 지정 결정한다. (파일 시스템 컴파일 방법 - 부록 A 참조, 파일 시스템 생성 방법 - 부록 F 참조)

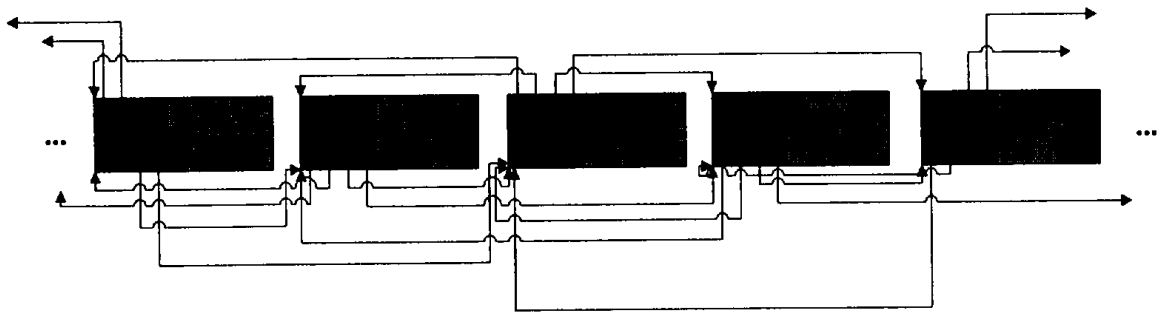
또, PMFS는 4MB, 8MB 또는 16MB 크기로 관리되는 Chunk 블록을 사용한다. Chunk 블록은 Chunk에 대한 정보를 가진 Chunk 헤더와 실제적인 데이터로 구성된다. Chunk 헤더는 Chunk의 첫번째 섹터에 저장되며 유지하는 데이터는 <표 3>과 같다.

<표 3> Chunk 헤더 구조

START_CODE	Chunk의 시작을 알리는 코드
ChunkNumber	Chunk의 물리적인 Chunk번호
Channel	이 Chunk를 사용하는 채널 번호
PrevChunk	같은 채널의 앞 Chunk 번호
NextChunk	같은 채널의 다음 Chunk 번호
PPrevChunk	같은 채널의 앞 앞 Chunk 번호
NNextChunk	같은 채널의 다음 다음 Chunk 번호
SeqNumber	이 채널에서의 일련 번호
Offset	실제 데이터가 시작되는 위치
Length	기록된 데이터의 크기
CheckSum	Chunk 헤더 체크 섬

실제 각 Chunk의 데이터는 두 번째 디스크 섹터에서 시작하며 하나의 Chunk에 기록할 수 있는 최대 데이터 양은 Chunk가 16MB인 경우, 16MB - 512B = 16,776,704 바이트이다.

각각의 Chunk들은 Chunk 헤더의 PrevChunk, NextChunk, PPrevChunk, NNextChunk 들에 의해 (그림 6)와 같이 이중 링크드 리스트로 유지된다.



(그림 6) Chunk들의 이중 링크드 리스트 연결

2.3 대용량의 연속 블록 크기 구조

PMFS는 4MB, 8MB, 16MB 크기의 연속 블록 크기를 지원한다. EXT2는 파일 시스템 생성 시 블록의 크기를 1KB, 2KB, 4KB 가운데 하나로 설정할 수 있다. 그러나 이같이 상대적으로 작은 블록 크기는 대형 멀티미디어 파일을 저장할 때 파일이 많은 블록에 디스크 상에 흩어져 디스크 탐색 시간을 증가시키는 원인이 되었다. PMFS는 대용량의 멀티미디어 파일을 효과적으로 저장하기 위해 Chunk를 4MB, 8MB, 16MB 크기로 설정할 수 있도록 설계되었으며 이를 통해 보다 많은 데이터를 연속적으로 유지하여 데이터의 분산을 막았다. 또한 이 같은 대용량의 단위 데이터 크기 구조는 기존 파일 시스템보다 메타 데이터 수와 디스크 I/O 수를 크게 줄여 시스템의 성능 향상을 가져온다.

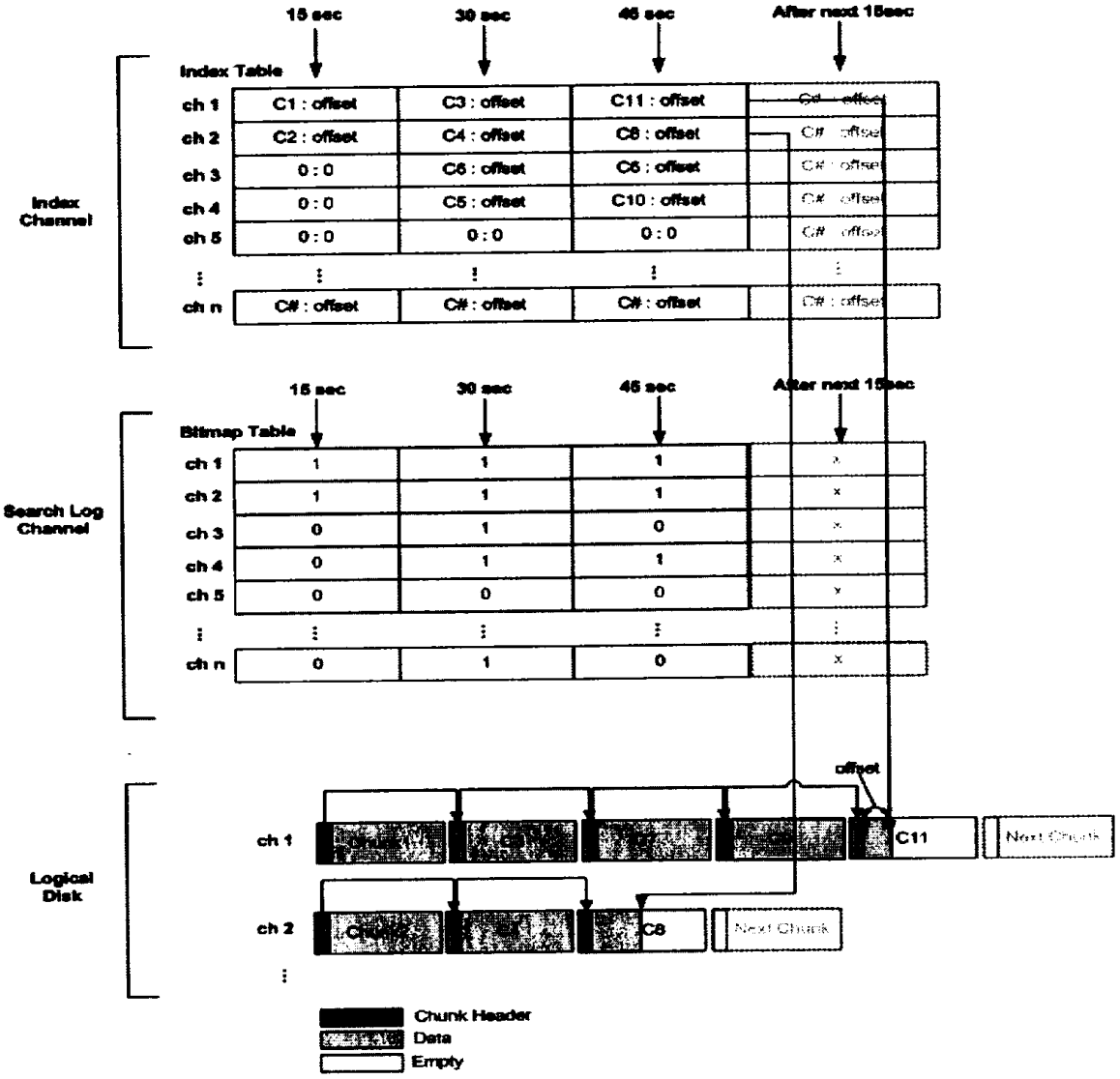
하지만 한 블록 크기를 4MB, 8MB, 16MB의 대형 블록 단위(Chunk)로 유지하면 내부 단편화 문제가 커지게 된다. 만일 10KB의 작은 크기의 데이터를 저장할 경우 나머지 15MB의 공간을 사용할 수 없게 되는 매우 심각한 문제가 발생한다. 그러나 개인용 멀티미디어 시스템의 경우 불과 몇 초 만에 수백 메가 바이트를 디스크에 순차적으로 기록하기 때문에 PMFS의 대부분의 Chunk는 꽉 차게 되며 단편화 문제는 거의 발생하지 않는다.

PMFS의 Chunk 크기는 파일 시스템을 생성 시 설정할 수 있으며 이후에는 임의로 바꾸는 것이 불가능하다. PMFS를 위한 성능측정에서 실제 Chunk 크기를 변경하여 실험을 실시하였다.

2.4 PMFS의 인덱스 구조

PMFS는 유닉스 계열 파일 시스템과 달리 아이노드를 통한 트리 구조를 사용하지 않고 (그림 7)과 같이 매우 단순한 인덱스 구조를 가진다. 인덱스 구조를 단순화한 이유는 아이노드를 이용한 트리 구조를 사용할 경우 많은 양의 메타 데이터를 유지해야 할 뿐 아니라 데이터를 순차적으로

읽는 경우에도 오버헤드를 발생시키기 때문이다. 실제 인덱스 구조는 응용 프로그램에 따라 다르게 만들 수 있으나 그림에서는 매 15초마다 각 채널의 데이터 위치를 기록하는 DVR의 경우를 묘사한 것이다.



(그림 7) PMFS의 디스크 데이터 배치

PMFS는 순차 인덱싱과 임의 위치 인덱싱 모두 가능하다. 순차 인덱싱은 (그림 5)에서 보듯이 *슈퍼 블록 --> 해당 채널 --> 첫번째 Chunk* 로 접근하여 이후 Chunk를 순차적으로 엑세스하는 것으로 이루어진다. 반면 임의 위치 인덱싱을 위해서는 별도의 두개의 채널을 가지고 있다. 그 중 인덱스 채널은 각 채널 별로 데이터의 위치를 Chunk 번호와 옵셋을 사용

하여 유지한다. 여기서 Chunk 번호는 디스크 상에서의 물리적 Chunk 번호이며 옵셋은 Chunk 내에서 데이터의 위치를 지정한다. 또 다른 한 채널은 인덱스 구간 ((그림 7)에서 15초) 동안 각 채널에 데이터가 기록되었는지를 비트맵으로 나타낸다.

만일 데이터가 쓰였다면 비트맵 필드는 1이 되며, 데이터가 쓰여지지 않았다면 0이 된다. 응용 프로그램은 인덱스 채널과 Search Log 채널을 열고 주기적으로 디스크에 인덱스 정보를 기록한다. 따라서 응용 프로그램은 채널 정보와 상대적 시간 정보를 가지고 원하는 데이터를 엑세스할 수 있다. 인덱스 채널을 사용하지 않으면 임의 위치 엑세스가 불가능하게 되지만 각 채널에 대한 순차 엑세스는 아직도 가능하다. 읽기는 임의, 순차 방식을 모두 지원하지만 쓰기의 경우 PMFS에서는 순차 쓰기만 지원한다.

인덱스 채널과 Search Log 채널의 기록 간격은 파일 시스템을 생성 시 사용자가 결정한다

2.5 순차 블록을 통한 순차 접근 구조

PMFS는 블록 그룹을 사용하지 않는다. 대신 순차적 연산을 위해 16MB(또는 4MB, 8MB)크기의 Chunk에 데이터를 순차적으로 기록한다. 따라서 디스크 헤드는 대용량의 Chunk를 순차 방향으로 접근하게 되며 디스크 탐색 시간은 현저하게 줄어들게 된다.

2.6 데이터 무결성 보장

PMFS는 데이터를 버퍼 캐쉬를 통하지 않고 디스크에 바로 기록하는 방법을 통해 불시의 전원 차단에도 높은 데이터 안정성을 보인다. 가전용으로 만들어지는 많은 멀티미디어 시스템은 사용자가 불시에 전원을 끄는 경우를 빈번하게 당하게 될 가능성이 높으며, 버퍼 캐쉬를 사용하는 경우 전원 차단과 함께 미처 디스크에 기록되지 못한 데이터를 잃게 되어 파일 시스템이 깨지게 된다. PMFS는 데이터를 디스크에 기록할 때 버퍼 캐시

를 통하지 않고 바로 디스크에 쓰도록 하여 최근의 데이터를 디스크에 저장하며, 간단한 인덱스 구조 때문에 불시의 전원 차단 후에 필요한 파일 시스템 검사 시간이 매우 짧다.

2.7 미리 읽기

PMFS는 읽기 속도를 높이고자 효과적인 미리 읽기(Read-ahead)를 수행한다. 멀티미디어 시스템은 읽기 위주의 연산을 수행한다. 이는 디스크에 쓰는 속도보다 디스크로부터 읽는 속도가 더 중요함을 말해준다. 따라서 PMFS는 이를 위해 미리 읽기를 기존 파일 시스템보다 효과적으로 수행함으로써 읽기 속도가 빨라졌다. PMFS의 경우 디스크로부터 읽기를 수행하면 32KB 데이터를 읽을 때마다 최대 16*32KB 양의 데이터를 읽기 버퍼에 유지한다. 따라서 일단 읽기를 수행하면 다음의 읽기를 위해 이미 충분한 데이터를 버퍼에 유지하게 되므로 읽는 시간이 획기적으로 줄어들게 된다.

2.8 PMFS의 API

PMFS는 자체 시스템 콜 인터페이스를 제공한다. 응용 프로그램은 POSIX와 거의 유사한 PMFS 자체 API 라이브러리를 통해 PMFS에 접근하여 파일에 대한 연산을 수행할 수 있다. 이 방법은 응용 프로그램의 호환성 면에서는 단점일 수 있으나, 범용 시스템이 아닌 개인용 멀티미디어 시스템에서는 크게 문제되지 않는다. <표 4>는 PMFS가 제공하는 중요 API를 보여 준다. (API에 대한 자세한 내용은 부록 C 참조)

<표 4> PMFS의 중요 API

API 함수 이름	채널	모드	기능
pmfs_open()	All		채널 읽기/쓰기를 위한 디스크립터 할당
pmfs_close()	All		디스크립터 반환
pmfs_read()	All	READ	채널에서 데이터 읽기
pmfs_write	All	WRITE	채널에 데이터 쓰기
pmfs_locate_index()	Index	READ	시간을 입력으로 인덱스 정보 위치 찾기
pmfs_iseek()	All	READ	인덱스 정보에 의한 채널 위치 찾기
pmfs_lseek()	All	READ	채널의 읽기/쓰기 위치 이동
pmfs_seekend()	Stream	READ	채널 데이터 끝으로 읽기 위치 이동
pmfs_tell()	All	All	채널의 현재 읽기/쓰기 위치 알아내기
pmfs_set_searchlog()			Search Log 정보 기록
pmfs_mkfs()			하드 디스크에 PMFS 파일 시스템을 만들기
pmfs_stat()			PMFS 파일시스템 상태 얻기
pmfs_sync()			메모리와 Disk의 모든 정보 일치시키기
pmfs_recycle()			파일 시스템의 예전 데이터를 지우기
pmfs_set_blocking()	All	WRITE	채널을 Blocking 쓰기 모드로 바꾸기
pmfs_set_nonblocking()	All	WRITE	채널을 Non Blocking 쓰기 모드로 바꾸기
pmfs_check_error()			가장 최근에 발생한 DISK error 확인

모든 함수는 성공하면 0 또는 양의 정수를 return 하며, 에러가 난 경우에는 음의 정수인 에러 코드를 리턴한다. (에러 코드에 대한 자세한 내용은 부록 B 를 참조)

IV. PMFS의 성능 평가 실험

본 실험에서는 EXT2, EXT3을 PMFS와의 성능 평가 비교 대상으로 선정하였으며 성능 평가 기준은 시험 프로그램에 대한 응답 시간으로 하였다.

1. 실험 환경

성능 평가를 실시할 시스템의 환경은 <표 5>와 같다.

<표 5> 시스템 환경

CPU	MIPS 4Kc (300Mhz)
Memory	32MB
IDE Interface	DMA 66
Linux Kernel	2.4.18

2. 파일 시스템 부하 설정

파일 시스템의 성능을 측정하기 위해 사용된 부하는 <표 6>와 같다.

<표 6> 성능 시험 용 부하

버퍼 크기	32KB
반복 회수	320(회)
단일 쓰레드 W/R 크기 (버퍼크기) * (Loop count)	10MB
쓰레드 수	1개, 4개, 8개

실제 쓰여지는 데이터는 메모리 상에 존재하는 32KB의 데이터이다. 테스트 프로그램은 쓰레드 수를 1개, 4개, 8개로 증가시키며 테스트를 수행

하며 버퍼 크기(32KB)만큼 해당 파일(PMFS의 경우는 해당 채널)에 쓰기, 읽기를 수행하게 된다. 총 쓰기/읽기 되는 데이터 크기는 쓰레드 수에 비례하여 10MB, 40MB, 80MB가 된다.

3. 테스트 프로그램 구현

테스트 프로그램은 멀티미디어 파일에 대한 성향을 잘 반영하고 있어야 한다. 개인용 멀티미디어 시스템의 경우 주로 순차적인 쓰기, 읽기 연산이 대부분이기 때문에 테스트 프로그램도 이와 같은 기능을 갖도록 구현되었다. <표 7>은 테스트 프로그램의 동작 방법에 대해 기록하고 있다.

<표 7> 테스트 프로그램 동작 방법

쓰기	*. EXT2, EXT3 1. 버퍼 크기만큼 메모리를 할당받아 채운다. 2. 각 쓰레드는 해당 파티션에 자신의 파일을 연다. 3. 각 쓰레드는 반복 회수만큼 돌며 버퍼에 있는 데이터를 해당 파일에 버퍼 크기씩 쓴다.
	*. PMFS 1. 버퍼 크기만큼 메모리를 할당받아 채운다. 2. 각 쓰레드는 해당 채널을 연다. 3. 각 쓰레드는 반복 회수만큼 돌며 버퍼에 있는 데이터를 해당 채널에 버퍼 크기씩 쓴다.
읽기	*. EXT2, EXT3 1. 버퍼 크기만큼 메모리를 할당 받는다. 2. 각 쓰레드는 해당 파티션에 자신의 파일을 연다. 3. 각 쓰레드는 반복 회수만큼 돌며 해당 파일로부터 버퍼 크기씩 읽어 버퍼에 저장한다.
	*. PMFS 1. 버퍼 크기만큼 메모리를 할당 받는다. 2. 각 쓰레드는 해당 채널을 연다. 3. 각 쓰레드는 반복 회수만큼 돌며 해당 채널로부터 버퍼 크기씩 읽어 버퍼에 저장한다.

4. 측정 방법

성능 측정을 수행할 때 시간 측정 방법과 버퍼 캐쉬의 효과를 없애는 전략이 고려되어야 한다. 특히, 버퍼 캐쉬를 통해 디스크 I/O가 일어나면 파일 시스템에 대한 정확한 성능 평가가 이루어질 수 없기 때문에 이에 대한 선행 처리를 해주는 것이 필요하다.

본 실험에서는 시간 측정을 위해 사용한 gettimeofday()는 리눅스 커널에서 제공하고 있는 함수로 마이크로 초 단위까지 측정이 가능하다. 측정 방법은 이 함수를 디스크 I/O를 수행하는 쓰레드들의 생성과 소멸 앞 뒤에 추가하여 쓰레드의 시작과 끝의 시간 차를 구하였다

버퍼 캐쉬의 효과를 없애는 일반적인 방법으로는 시스템에 메인 메모리보다 큰 파일을 쓰는 방법과 O_DIRECT를 사용하는 방법, 시스템을 재부팅하는 방법 등이 있다[Bryant외, 2002]. 본 실험에서는 실험 전 시스템을 재부팅하는 것을 통해 버퍼 캐쉬의 효과를 없앴다. 실험 순서는 <표 8>과 같다

<표 8> 성능 평가 실험 순서

1차				2차			
PMFS Chunk 크기별 평가				파일 시스템별 평가			
4MB	W	R	W/R	EXT2	W	R	W/R
8MB	W	R	W/R	EXT3	W	R	W/R
16MB	W	R	W/R	PMFS	W	R	W/R

성능 평가를 위한 실험은 두 단계로 이루어졌다. 1 단계 실험은 PMFS의 Chunk 크기에 따른 성능 변화를 평가하여 가장 높은 성능을 보이는 Chunk 크기를 결정하는 실험으로 Chunk 크기를 4MB, 8MB, 16MB로 변경하며 각각 쓰기, 읽기, 쓰기/읽기 동시 수행에 대한 성능 평가를 실시하

였다. 2 단계 실험은 파일 시스템 별 평가로 1 단계 실험에서 결정된 Chunk 크기를 적용한 PMFS와 EXT2, EXT3과의 성능 비교 평가를 실시하였다.

이때, 각각의 실험들에 대해 쓰레드 수를 1개, 4개 8개로 변경하며 성능을 측정하였으며, 쓰레드 수가 1개인 경우 쓰기/읽기 동시수행에 대한 평가를 수행하지 않았다. 4개, 8개의 쓰레드 수행에 있어서는 쓰기/읽기의 쓰레드 비율을 1:4로 잡았는데 이는 멀티미디어 시스템이 읽기 위주라는 특징을 반영한 것이다.

V. 결과 분석 및 평가

본 장에서는 앞서 정의된 시험용 부하를 이용해 성능 평가를 실시하였다. 평가는 1 단계, 2 단계로 나누어 실시하였으며 1 단계 실험은 PMFS의 Chunk 크기 별 평가를, 2 단계 실험은 1 단계 실험 결과로 알게 된 최적 Chunk 크기를 반영한 PMFS와 다른 리눅스 파일 시스템과의 평가를 실시하였다. 2 단계 성능 평가에서는 EXT2와 EXT3을 PMFS와의 성능 비교 대상으로 선정하였으며 성능 평가 기준은 시험 프로그램에 대한 응답 시간으로 하였다.

1. PMFS Chunk 크기 별 성능 평가

PMFS에 적용할 최적의 Chunk 크기를 결정하기 위해 4MB, 8MB, 16MB로 Chunk 크기를 변경하며 쓰기, 읽기, 쓰기/읽기 동시수행에 대한 성능을 측정하였다.

쓰기에 대한 성능평가 결과는 <표 9>과 같다.

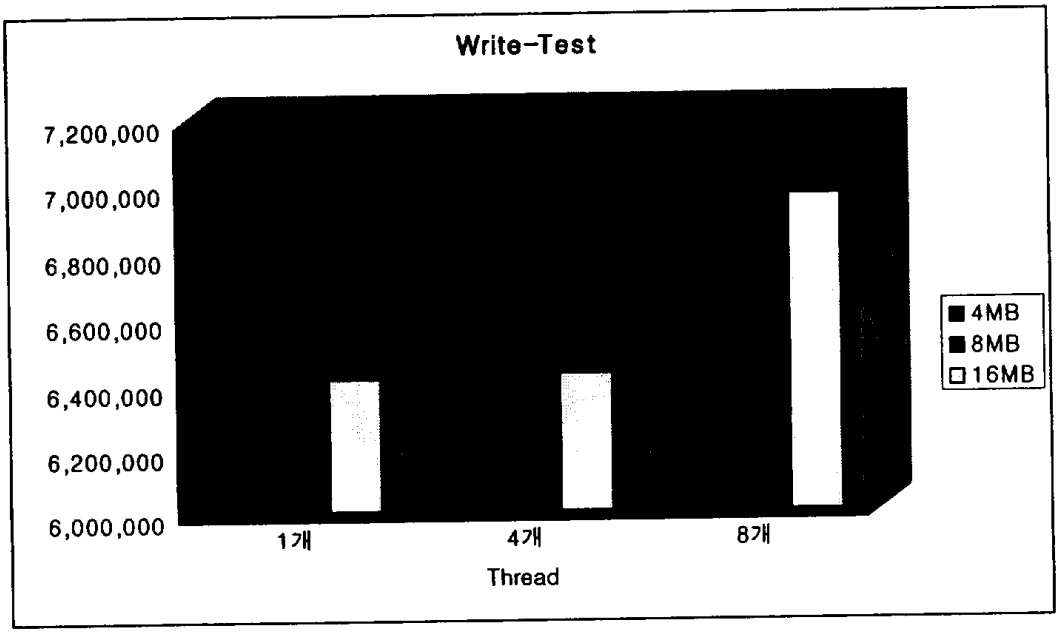
<표 9> PMFS의 Chunk 크기 별 쓰기 성능측정 결과

(단위: Microsecond)

Thread / Chunk	1개	4개	8개
4MB	6,416,667	6,443,333	7,076,667
8MB	6,406,667	6,436,667	7,036,667
16MB	6,406,667	6,420,000	6,963,333

쓰기의 경우 쓰레드 수가 1개, 4개일 경우 0.3%미만으로 세 경우 거의 차이가 없어 보인다. 그러나 쓰레드 8개를 동시에 돌렸을 경우 16MB가 4MB, 8MB보다 각각 1.6%, 1% 높게 나타났다. 이는 Chunk 크기가 크면 클수록 한 Chunk 당 기록할 수 있는 양이 증가하여 보다 적은 Chunk를

사용하므로 free Chunk를 가져오는데 거리는 시간을 줄일 수 있기 때문이다. (그림 8)는 위의 결과를 도식화하여 보여준다.



(그림 8) PMFS의 Chunk 크기 별 쓰기 성능 측정 결과

읽기에 대한 성능평가 결과는 <표 10>과 같다.

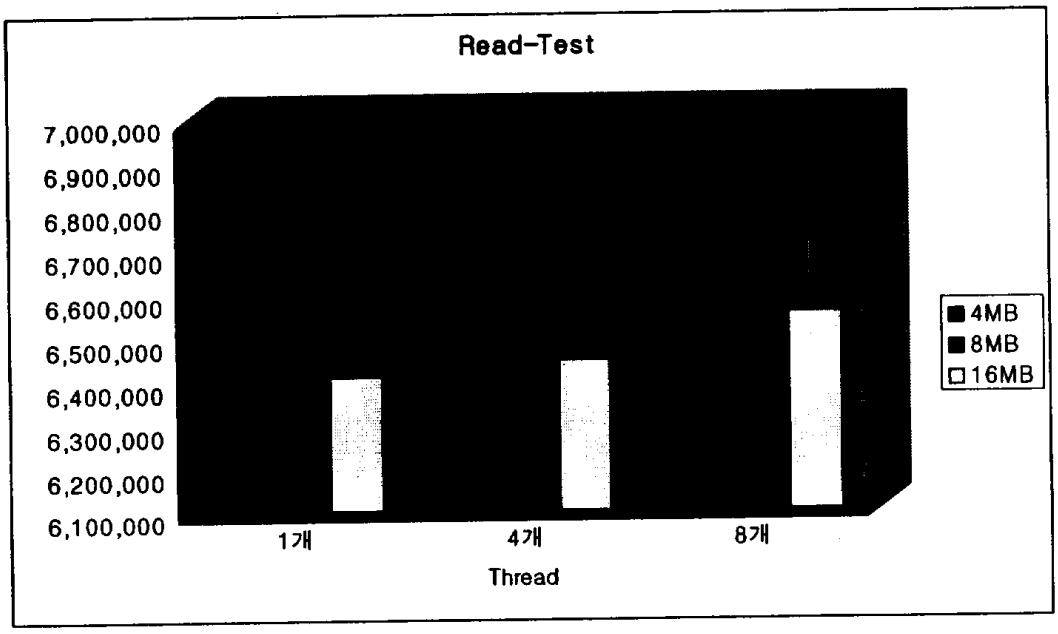
<표 10> PMFS의 Chunk 크기 별 읽기 성능측정 결과

(단위: Microsecond)

Thread / Chunk	1개	4개	8개
4MB	6,406,667	6,476,667	6,950,000
8MB	6,406,667	6,473,333	6,720,000
16MB	6,410,000	6,446,667	6,553,333

읽기의 경우 쓰레드 수가 1개, 4개일 경우 쓰기와 마찬가지로 거의 차이가 없어 보인다. 그러나 쓰레드 수가 8개의 경우 16MB가 4MB, 8MB보다 각각 5.71%, 2.48% 높게 나타났다. 이는 미리 읽기 정책이 같다해도 사이즈가 큰 Chunk 일수록 보다 많은 데이터를 읽게 되므로 상대적으로 적

은 수의 Chunk에 접근하게 된다. 따라서 Chunk를 찾는데 드는 시간이 줄어들게 되기 때문이다. (그림 9)는 위의 결과를 도식화하여 나타내 주고 있다.



(그림 9) PMFS의 Chunk 크기 별 읽기 성능 측정 결과

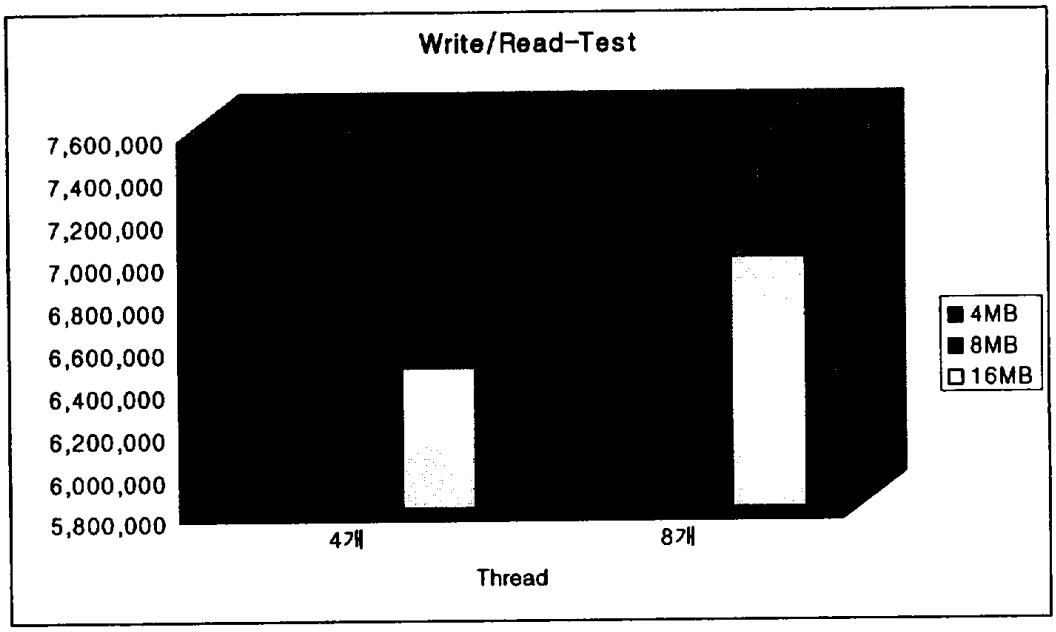
쓰기/읽기 동시수행에 대한 성능평가 결과는 <표 11>과 같다.

<표 11> PMFS의 Chunk 크기별 쓰기/읽기 성능측정 결과

(단위: Microsecond)

Chunk \ Thread	4개	8개
4MB	6,470,000	7,463,333
8MB	6,466,667	7,326,667
16MB	6,456,667	6,973,333

쓰기/읽기 동시 수행의 경우 쓰레드 수가 4개일 경우 큰 차이를 보이지 않는다. 그러나 8개의 경우 16MB가 4MB, 8MB보다 각각 6.57%, 4.82% 높게 나타났다. (그림 10)는 위의 결과를 도식화하여 나타내 주고 있다.



(그림 10) PMFS의 Chunk 크기 별 쓰기/읽기 성능측정 결과

이 같은 결과는 Chunk 크기가 크면 클수록 멀티미디어 데이터에 대한 처리 성능이 높아지는 것을 보여주며 동시에 현재 가장 높은 성은을 보인 PMFS의 Chunk 크기는 16MB임을 나타낸다.

2. 파일 시스템별 성능 평가

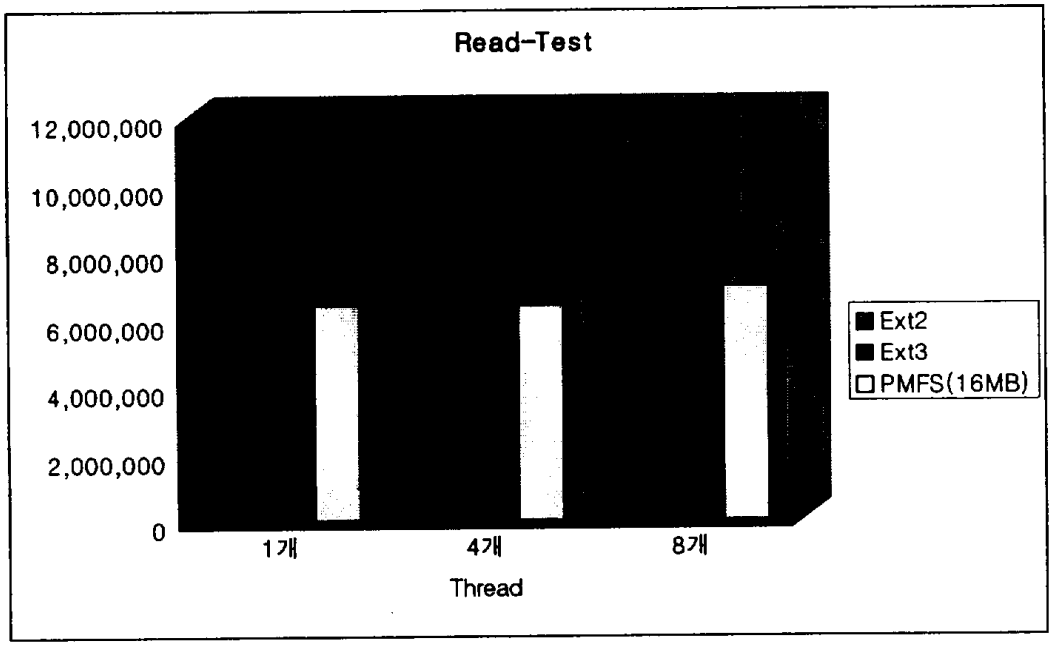
여기서는 5.1의 실험 결과 가장 성능이 우수한 16MB(Chunk 크기) PMFS와 EXT2, EXT3의 성능 평가를 실시하였다.

쓰기에 대한 성능평가 결과는 <표 12>과 같다.

<표 12>파일 시스템 별 쓰기 성능측정 결과 (단위: Microsecond)

FS \ Thread	1개	4개	8개
EXT2	6,413,333	7,590,000	11,556,667
EXT3	6,413,333	7,750,000	11,736,667
PMFS(16MB)	6,406,667	6,420,000	6,963,333

쓰기의 경우 쓰레드 수가 1개일 경우 차이가 없으며 4개의 경우 PMFS가 EXT2, EXT3 보다 각각 15.42%, 17.16% 성능이 높은 것으로 나타났다. 또한 8개의 경우 각각 39.75%, 40.67% 성능이 높은 것으로 나타났다. 쓰기 성능이 높은 이유는 PMFS가 EXT2, EXT3 보다 훨씬 적은 양의 메타 데이터를 쓰므로 디스크 헤드의 움직임으로 인한 성능 저하가 적기 때문이다. 또, 쓰레드 수가 증가할수록 성능 차이가 크게 나타나는 것은 EXT2, EXT3의 경우 쓰레드 1개일 때 데이터를 비교적 근접한 블록에 기록하나 프로세스가 많아 지고, 데이터가 많아지면서 데이터가 디스크 상에 분산 되어 디스크 헤드의 이동 시간이 증가하기 때문이다. 반면 PMFS의 경우 16MB Chunk에 데이터를 순차적으로 기록하므로 데이터 분산이 줄어들기 때문에 높은 성능을 보인다. (그림 11)는 위의 결과를 도식화하여 보여준다.



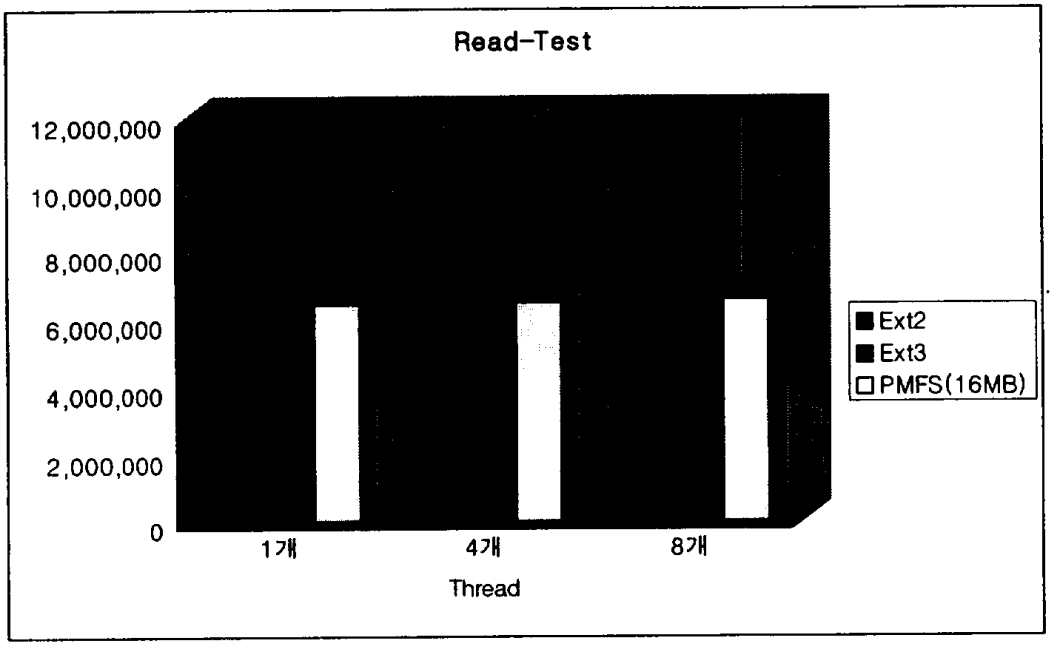
(그림 11) 파일 시스템 별 쓰기 성능 측정 결과

읽기에 대한 성능평가 결과는 <표 13>과 같다.

<표 13>파일 시스템별 읽기 성능측정 결과 (단위: Microsecond)

FS \ Thread	1개	4개	8개
EXT2	6,536,667	6,610,000	10,316,667
EXT3	6,533,333	6,583,333	11,570,000
PMFS(16MB)	6,410,000	6,446,667	6,553,333

읽기의 경우 쓰레드 수가 1개일 경우 PMFS가 EXT2, EXT3보다 1.94%, 1.89%, 4개일 경우 2.47%, 2.08%, 8개일 경우 36.48%, 43.36% 높은 성능을 보였다. PMFS의 읽기 성능이 높은 이유는 EXT2와 EXT3은 작은 블록들을 트리 구조를 통해 접근하지만 PMFS는 훨씬 큰 순차 블록을 엑세스하여 디스크 검색 속도를 현저히 감소시키기 때문이다. 또한, PMFS가 미리 읽기를 보다 많이 효과적으로 하여 디스크 I/O 수를 감소시키기 때문이다. (그림 12)는 위의 결과를 도식화한 것이다.



(그림 12) 파일 시스템 별 읽기 성능 측정 결과

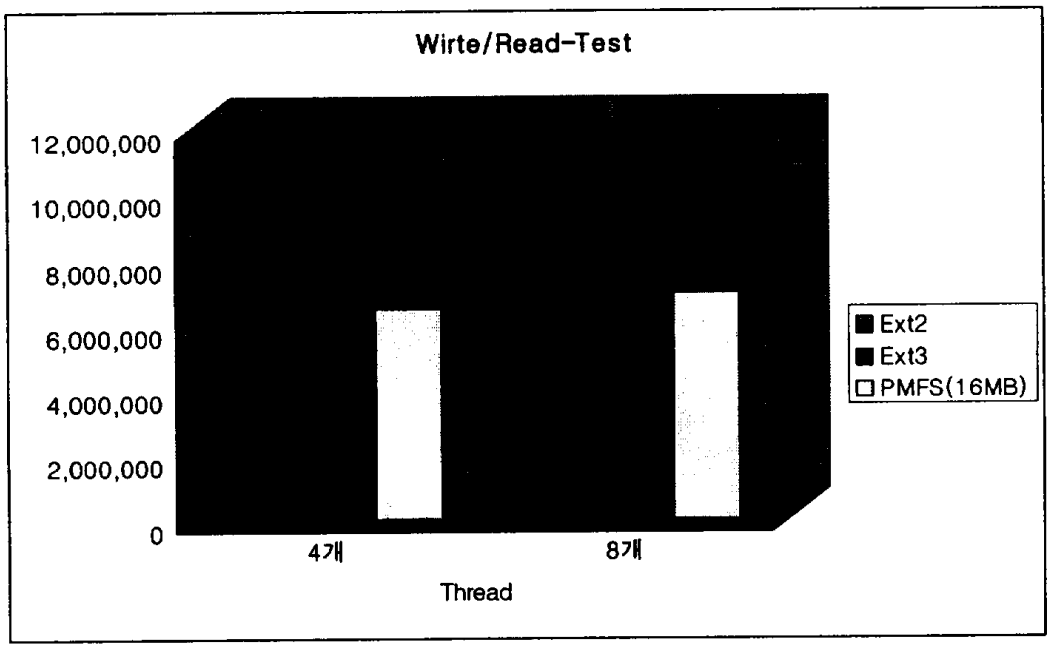
쓰기/읽기 동시 수행에 대한 성능 측정 결과는 <표 14>과 같다.

<표 14> 파일 시스템 별 쓰기/읽기 성능 측정 결과

(단위: Microsecond)

FS \ Thread	4개	8개
EXT2	6,576,667	9,690,000
EXT3	6,550,000	10,346,667
PMFS(16MB)	6,456,667	6,973,333

쓰기/읽기 동시 수행의 경우 쓰레드가 4개일 경우 PMFS가 EXT2, EXT3보다 각각 1.82%,1.42%, 8개일 경우 28.04%, 32.60%를 나타냈다. (그림 13)는 위의 결과를 도식화한 것이다.



(그림 13) 파일 시스템 별 쓰기/읽기 성능 측정 결과

VI. 결론

기존 리눅스 파일 시스템은 대용량 파일 처리 미흡, 느린 읽기 속도, 부적절한 인덱스 구조, 데이터 무결성 문제로 인하여 대형 멀티미디어 파일이 저장되고 재생되는 개인용 멀티미디어 시스템을 지원하기 어려웠다. 본 연구에서는 위와 같은 문제점을 해결할 수 있는 새로운 파일 시스템인 PMFS를 설계하고 구현하였다.

PMFS는 데이터의 순차 관리, 적은 메타 데이터, 채널별 데이터 관리, 대용량 Chunk의 사용, 효과적인 미리 읽기 등을 통해 개인용 멀티미디어 시스템의 성능과 안정성을 확보하였다. 또한 성능 평가를 통해 PMFS의 최적 Chunk 크기가 16MB이며 PMFS가 EXT2, EXT3보다 안정성, 성능 면에서 뛰어남을 검증하였고 이를 통해 PMFS가 멀티미디어 파일 시스템으로서 적합함을 보였다. 이 같은 결과는 멀티미디어 시스템의 경우 PMFS를 사용하는 것이 시스템의 성능과 안정성을 크게 높일 수 있는 대안임을 말해준다.

PMFS의 구현은 앞으로도 지속적인 발전이 예상되는 멀티미디어 시스템을 위한 파일 시스템의 구현을 위한 좋은 기초 자료가 되며, 리눅스나 그 밖의 운영 체제를 사용하는 멀티미디어 상품에 쉽게 적용될 수 있다.

앞으로 응용프로그램에 우선순위를 설정하고 높은 우선순위에 대한 요청을 먼저 수행하는 서비스 품질(QoS)을 위한 고려가 파일 시스템에 추가되어야 할 것이다.

참 고 문 헌

권수호, Linux Kernel Programming Bile 2nd Edition, 글로벌, 2002

권우일, 윤미현, 이동준, 장재혁, 양승민, DVR 시스템을 위한 저널링 파일 시스템의 성능평가, 한국정보과학회지

원유집, 박진연, "정보가전용 멀티미디어 파일 시스템 기술" 멀티미디어와 정보화사회, 한국과학기술진흥재단, http://211.40.179.13/book_file/ke28/ke028-index.htm

Sweeney, A., Doucette, D., Hu, W., Anderson, C. Nishimoto, M., and Peck G., "Scalability in the XFS File System", 1996 USENIX conference, 1996 http://oss.sgi.com/projects/xfs/papers/xfs_usenix/#fn1

Czezatke, Christian and Ertl, M.A., "A Log-Structured Filesystem for Linux" 2000 USENUX Annual Technical Conference, 2000

Tweedie, Stephen, "EXT3, Journaling Filesystem", the Ottawa Linux Symposium, html document of OLS, 20 July, 2000 http://olstrans.sourceforge.net/release/OLS2000-EXT3/OLS2000-EXT3.html

Bar, Moshe, Linux File Systems, McGraw-Hill Companies, 2001

Card, R., Ts'o, T., and Tweedie, S., "Design and Implementation of the Second Extended Filesystem", First Dutch International Symposium on Linux, ISBN 90-367-0385-9, 1994 http://e2fsprogs.sourceforge.net/EXT2intro.html

Bryant, R., Forester, Ruth and Hawkes, John, "Filesystem Performance and Scalability in Linux 2.4.17," 2002 USENIX Annual Technical Conference, 2002

Journaled File System Technology for Linux, html document of IBM, http://oss.software.ibm.com/jfs/

XFS html document of SGI, http://oss.sgi.com/projects/xfs/

Robbins, Daniel, "고급 파일 시스템 개발자 가이드 , Part 7 EXT3", IBM, http://www-903.ibm.com/developerworks/kr/linux/library/l-fs7.html

부 록

[부 록 A] 컴파일/링크/실행, /proc/pmfs 파일 시스템 내용

PMFS 파일 시스템 이용을 위해서는

#include <pmfsif.h>

를 소스에 추가하고 컴파일 후 pmfs_lib.o를 링크한다.

실제 PMFS 파일 시스템은 커널 내부에서 동작하며, 정상적으로 실행된 경우
"pmfsd"라는 이름의 커널 데몬이 동작하는 것을 "ps"로 확인할 수 있으며

/proc/pmfs 디렉토리 아래 다음과 같은 파일이 생겨 정보를 알 수 있다.

/proc/pmfs/disk : 파일 시스템이 이용하고 있는/발견한 HARD DISK 상황첫 줄 첫 word에 숫자로 디스크 개수 표시
/proc/pmfs/channel : 각 channel의 사용, chunk, buffer 상황
/proc/pmfs/perf : PMFS의 성능 (User< >커널, DISK Thruput)
/proc/pmfs/stat : PMFS 통계
/proc/pmfs/user : 현재 열린 descriptor 들의 사용 채널과 현재 Chunk, offset 값

"stat" 명령으로 현재 PMFS의 Super Block 상태를 읽을 수 있다. (부록 F 참조)

[부 록 B] PMFS의 에러 선언

```
#define PMFS_ENOMEM          -1      /* No Memory */
#define PMFS_ENOSPACE        -2      /* No Descriptor Space */
#define PMFS_EWRITER_ALREADY -3 /* There is another Writer */
#define PMFS_EINTERNAL       -4  /* PMFS File System Internal Error */
#define PMFS_EPERM           -5      /* Permission Error */
#define PMFS_ENOINDEX        -6      /* No Index exist */
#define PMFS_EBADD           -7      /* Bad Descriptor */
#define PMFS_EWOULDBLOCK     -8      /* Write would block */
#define PMFS_ENEEDRECYCLE    -9      /* PMFS Need Recycle */
#define PMFS_EINVALID        -10     /* Invalid argument */
#define PMFS_EINUSE          -11     /* PMFS is in use */
#define PMFS_ENOPMFS         -12     /* No PMFS in System */
#define PMFS_EBROKEN         -13     /* PMFS File system Broken */
#define PMFS_ENODISK         -14     /* No Hard Disk Found */
#define PMFS_ENEEDFSCK       -15     /* Need File System Check */
#define PMFS_EALREADY_LOCKED -16 /* PMFS_FS Already Locked */
#define PMFS_EDISKACCESS        -17 /* PMFS met DISK Error */
```

[부 록 C] PMFS의 API 내부 동작

<table>
<tr><th colspan="3">pmfs_open()</th></tr>
<tr><td>기능</td><td colspan="2">PMFS 파일 시스템의 한 채널을 사용하기 위한 디스크립터를 할당 받는다.</td></tr>
<tr><td>사용법</td><td colspan="2">#include <pmfsif.h>
int pmfs_open(int channel, int mode);</td></tr>
<tr><td>인자</td><td colspan="2">mode : PMFS_CH_READ (읽기), PMFS_CH_WRITE (쓰기) 둘 중의 하나
channel : PMFS_CH_SUPER에서 PMFS_CH_LOG 사이의 채널번호</td></tr>
<tr><td>설명</td><td colspan="2">원하는 채널을 읽기 또는 쓰기 용도로 열어 디스크립터를 리턴 값으로 받는다. 이 디스크립터는 모든 다른 PMFS API 함수에 이용된다.
한 채널에 대하여, 단 한번의 쓰기 모드 pmfs_open()이 허용되며, 읽기 모드의 pmfs_open()은 여러 번 가능하다. 즉, 한 채널에 대하여 여러 개의 읽는 프로세스/쓰레드가 허용되며, 쓰는 프로세스/쓰레드는 단 하나만 존재할 수 있다.
채널을 읽기 모드로 pmfs_open() 한 직후, 내부의 읽기 위치는 채널에 저장된 첫번째 데이터를 가리킨다. 채널을 쓰기 모드로 pmfs_open() 하면 내부의 쓰기 위치는 채널의 마지막 데이터 다음 위치를 가리키며, 채널에 대한 쓰기 모드는 Non Blocking 모드로 설정된다.
인덱스 채널은 읽기 모드로만 pmfs_open()이 가능하다.
인덱스 채널에 대한 쓰기는 커널 내부에서 이루어진다.</td></tr>
<tr><td rowspan="2">리턴값</td><td>성공</td><td>0 보다 같거나 큰 디스크립터</td></tr>
<tr><td>실패</td><td>PMFS_ENOPMFS : PMFS 파일 시스템이 없는 경우
PMFS_ENOMEM : 커널 내부에 버퍼가 부족하여 에러가 난 경우
PMFS_EPERM : 인덱스 채널에 쓰기 요청을 한 경우
PMFS_ENOSPACE : 디스크립터 영역이 모자라는 경우
PMFS_EWRITER_ALREADY : 이미 이 채널이 쓰기 모드로 열린 경우
PMFS_EINTERNAL : PMFS 파일 시스템 내부에 문제가 발생한 경우
PMFS_EINVALID : channel 이나 mode가 잘못된 경우
PMFS_ENODISK : HDD가 없는 경우
PMFS_EDISKACCESS : 이전에 하드 디스크에서 에러가 난 경우</td></tr>
<tr><td>참고</td><td colspan="2">pmfs_locate_index() 시간에 의한 인덱스 채널 위치 찾아가기
pmfs_lseek() 채널에서의 논리적인 위치 이동하기
pmfs_set_blocking() 채널에 대한 쓰기 모드를 Blocking 모드로 바꾼다.</td></tr>
</table>

pmfs_close()		
기능	채널에 대한 사용을 마친다.	
사용법	#include <pmfsif.h> int pmfs_close(int dd);	
인자	dd : pmfs_open()에 의해 얻어진 채널 디스크립터	
설명	디스크립터 dd를 닫는다. pmfs_close()는 커널 내부에 dd가 사용하던 모든 자원을 반환한다. 또 pmfs_close() 결과 dd가 지정한 채널에 대하여 더 이상 사용하는 프로세스나 쓰레드가 없는 경우 그 채널에 할당된 자원도 반환된다.	
리턴값	성공	0
	실패	PMFS_EINTERNAL : PMFS 파일 시스템 내부에 문제가 발생한 경우 PMFS_EBADD : 디스크립터가 잘못 지정된 경우
참고		

<table>
<tr><th colspan="3">pmfs_read()</th></tr>
<tr><td>기능</td><td colspan="2">채널의 데이터를 읽는다.</td></tr>
<tr><td>사용법</td><td colspan="2">#include <pmfsif.h>
int pmfs_read(int dd, void *buffer, int count);</td></tr>
<tr><td>인자</td><td colspan="2">dd : pmfs_open()에 의해 얻어진 읽기 모드 채널 디스크립터
buffer : 디스크의 채널 데이터를 읽어올 미리 할당된 버퍼의 주소
count : 읽고자 하는 바이트 수</td></tr>
<tr><td>설명</td><td colspan="2">dd가 지정하는 채널의 현재 읽기 위치에서 데이터를 count가 지정한 바이트 수 만큼 읽어 buffer에 저장한다. pmfs_read()는 실제 읽은 데이터의 크기를 리턴하며, 에러가 없을 경우, 그 값은 언제나 0보다 크거나 같고 count 보다 작거나 같다. count 보다 리턴 값이 작은 경우는 채널에 더 이상 읽을 데이터가 없는 경우이다.
pmfs_read()는 읽을 데이터가 디스크에서 전송되는 동안이나, 데이터 커널의 메모리 영역이 모자라 버퍼 할당이 늦어지는 경우 블록될 수 있다. pmfs_read()는 Non Blocking 모드 I/O를 지원하지 않는다.
이 채널이 쓰기 모드로도 열려 있는 경우, 가장 최근에 쓰여진 데이터까지를 읽을 수 있다.
Recycle 등으로 인덱스 채널에서 현재 읽기 위치의 데이터가 삭제된 경우, PMFS_ENOINDEX로 리턴한다.</td></tr>
<tr><td rowspan="2">리턴값</td><td>성공</td><td>실제로 읽은 바이트 수 (0보다 같거나 크고 count 보다 작거나 같은)</td></tr>
<tr><td>실패</td><td>PMFS_ENOMEM : 커널 내부에 버퍼가 부족하여 에러가 난 경우
PMFS_EINTERNAL : PMFS 파일 시스템 내부에 문제가 발생한 경우
PMFS_EBADD : 디스크립터가 잘못 지정된 경우
PMFS_EPERM : dd가 쓰기 모드로 열린 경우
PMFS_EINVALID : buffer가 NULL이거나 잘못된 영역인 경우
PMFS_ENOINDEX : 인덱스 채널에서 읽으려는 위치가 없어진 경우
PMFS_EDISKACCESS : 이전에 하드 디스크에서 에러가 난 경우</td></tr>
<tr><td>참고</td><td colspan="2">pmfs_locate_index() 시간에 의한 인덱스 채널 위치 찾아가기
pmfs_lseek() 채널에서의 논리적인 위치 이동하기</td></tr>
</table>

<table>
<tr><th colspan="3">pmfs_write()</th></tr>
<tr><td>기능</td><td colspan="2">디스크의 해당 채널에 데이터를 쓴다.</td></tr>
<tr><td>사용법</td><td colspan="2">#include <pmfsif.h>
int pmfs_write(int dd, void *buffer, int count);</td></tr>
<tr><td>인자</td><td colspan="2">dd : pmfs_open()에 의해 얻어진 쓰기 모드의 채널 디스크립터
buffer : 디스크에 기록할 데이터를 저장하고 있는 버퍼의 주소
count : 쓰고자 하는 바이트 수</td></tr>
<tr><td>설명</td><td colspan="2">dd가 지정하는 채널의 현재 쓰기 위치에서부터 buffer의 데이터를 count가 지정하는 바이트 수 만큼 쓴다. pmfs_write()가 성공하면, 채널에 쓰여진 데이터의 크기를 리턴하여 언제나 count와 같다. 실패인 경우에는 한 바이트도 채널에 써지지 않으며 에러 번호를 가지고 리턴한다.
pmfs_write() 함수에서의 리턴은 buffer의 데이터가 디스크에 실제로 저장된 것을 의미하지 않는다. 이 함수는 buffer의 데이터를 모두 커널 영역에 복사한 뒤에 리턴하며 실제 디스크에 써지는 것은 잠시 뒤이다. 채널이 Blocking 쓰기 모드로 지정되면, pmfs_write() 때 커널의 메모리 영역이 모자라 버퍼 할당이 늦어지는 경우나 기타 여러 경우에 Block될 수 있다. 채널이 Non Blocking 쓰기 모드일 때는, 그 경우 Block 되지 않고 PMFS_EWOULDBLOCK 에러로 리턴하며, buffer의 데이터는 한 바이트로 쓰여지지 않는다. (쓰기 모드로 pmfs_open()을 하면 디폴트로 Non Blocking 쓰기 모드가 되며, pmfs_set_blocking() 함수로 채널을 Blocking 쓰기 모드로 변경할 수 있다.)</td></tr>
<tr><td rowspan="2">리턴값</td><td>성공</td><td>Count</td></tr>
<tr><td>실패</td><td>PMFS_EINTERNAL : PMFS 파일 시스템 내부에 문제가 발생한 경우
PMFS_EBADD : 디스크립터가 잘못 지정된 경우
PMFS_EWOULDBLOCK : 자원 부족으로 블록을 해야할 경우
PMFS_EPERM : dd가 읽기 모드로 열린 경우
PMFS_ENEEDRECYCLE : PMFS 파일 시스템의 용량이 부족함
PMFS_EINVALID : buffer가 NULL이거나 잘못된 영역을 가리키는 경우, 또는 size가 16MB 이상인 경우</td></tr>
<tr><td>참고</td><td colspan="2">pmfs_set_blocking() 쓰기 채널을 Blocking 모드로 설정한다.
pmfs_set_nonblocking() 쓰기 채널을 Non Blocking 모드로 설정한다.
pmfs_recycle() PMFS 파일 시스템의 가장 오래된 데이터를 삭제한다.</td></tr>
</table>

<table>
<tr><th colspan="3">pmfs_locate_index()</th></tr>
<tr><td>기능</td><td colspan="2">인덱스 채널의 읽기 위치를 지정된 시간의 인덱스 데이터 저장 위치로 이동한다.</td></tr>
<tr><td>사용법</td><td colspan="2">#include <pmfsif.h>
#include <time.h>
int pmfs_locate_index(int dd, time_t time);</td></tr>
<tr><td>인자</td><td colspan="2">dd : pmfs_open()에 의해 얻어진 읽기 모드 인덱스 채널의 디스크립터
time : 찾고자 하는 시간, (초 단위 00:00:00 UTC, 1970년 1월1일 기점)</td></tr>
<tr><td>설명</td><td colspan="2">pmfs_locate_index() 함수는 수퍼 인덱스 채널 (PMFS_CH_SUPER), 인덱스 채널 (PMFS_CH_INDEX), Search Log 인덱스 채널(PMFS_CH_SINDEX)에서 읽기 위치를 원하는 시간의 인덱스 데이터가 있는 곳으로 옮긴다. 이 함수는 인덱스 채널에 대해서만 동작하며 채널에서는 PMFS_EPERM 에러로 리턴한다.
time은 수퍼 인덱스의 경우 한 시간 단위, 인덱스와 Search Log 채널의 경우 15초 단위(0,15,30,45초)로 지정되어야 하며, 그렇지 않은 경우 라이브러리에서 time에 가장 가까운 과거의 한 시간, 15초 단위 시간으로 수정한다 (즉 x년x월x일x시x분17초가 지정되면 수퍼 인덱스의 경우 해당 연월일시 0분 0초, 인덱스와 Search Log 채널의 경우 해당 연월일시분15초로 변경된다.)
인덱스와 Search Log 인덱스에 대한 pmfs_locate_index()는 내부적으로 해당 시간에 대한 수퍼 인덱스 정보를 참조하여 처리된다.
PMFS 파일 시스템에서 모든 인덱스는 커널에 의해 자동으로 기록되며, pmfs_locate_index() 함수에서는 그 인덱스 정보를 시간을 인자로 읽게 된다.
이 함수에서 실패하는 경우, 채널에 대한 읽기 위치는 임의의 값이 된다.</td></tr>
<tr><td rowspan="2">리턴값</td><td>성공</td><td>0</td></tr>
<tr><td>실패</td><td>PMFS_EPERM : dd가 인덱스 채널이 아닌 경우
PMFS_ENOINDEX : time에 지정된 시간의 인덱스 데이터가 존재하지 않은 때 (recycle로 지워졌거나, time이 아직 오지 않은 미래인 경우).
PMFS_EINTERNAL : PMFS 파일 시스템 내부에 문제가 발생한 경우
PMFS_ENOSPACE : PMFS 파일 시스템 내부에 작업 공간이 부족한 경우
PMFS_EBADD : 디스크립터가 잘못 지정된 경우</td></tr>
<tr><td>참고</td><td colspan="2">pmfs_iseek() 채널에서의 물리적인 디스크 위치 찾아가기
pmfs_lseek() 채널에서의 논리적인 위치 찾아가기
pmfs_tell() 채널의 물리적인 읽기/쓰기 포인터 값 알아내기
시간 표현 변환 함수 들:
mktime() 년/월/일/시/분/초 형식의 현지 시간을 time_t 형식의 시간으로
localtime() time_t 형식의 시간을 년/월/일/시/분/초 형식의 현지 시간으로
gmtime() time_t 형식의 시간을 년/월/일/시/분/초 형식의 UTC 시간으로</td></tr>
</table>

<table>
<tr><th colspan="3">pmfs_iseek()</th></tr>
<tr><td>기능</td><td colspan="2">채널에서 읽기 위치를 데이터의 물리적인 위치로 이동한다.</td></tr>
<tr><td>사용법</td><td colspan="2">#include <pmfsif.h>
int pmfs_iseek(int dd, struct StreamIndex *position);</td></tr>
<tr><td>인자</td><td colspan="2">dd : pmfs_open()에 의해 얻어진 수퍼 인덱스 채널을 제외한 읽기 모드 채널의 디스크립터
position : StreamIndex 구조에 따른 채널 데이터의 물리적인 위치
(즉, 인덱스 채널을 읽어 얻어낸 값)</td></tr>
<tr><td>설명</td><td colspan="2">pmfs_iseek()는 dd가 지정하는 채널의 읽기 위치를 position이 지정하는 물리적인 위치로 옮기기 위해 사용한다.
pmfs_iseek() 함수를 이용하기 위해서는 먼저, 인덱스 정보 (수퍼 인덱스 PMFS_CH_SUPER, 또는 인덱스 PMFS_CH_INDEX)를 참조하여 dd가 지정하는 채널의 원하는 시간의 데이터 시작 위치인 인덱스 데이터(struct StreamIndex)를 얻어낸 뒤 (인덱스 채널에 대하여, pmfs_locate_index()를 하여 원하는 시간에 해당되는 위치를 찾아간 뒤, pmfs_read()를 이용하여 position 값을 얻는다.), dd가 지정하는 채널의 해당 물리적인 위치로 읽기 위치를 이동할 때 사용된다.
struct StreamIndex에는 PMFS 파일 시스템에서의 물리적인 Chunk 번호와 그 Chunk에서의 offset이 들어있다.</td></tr>
<tr><td rowspan="2">리턴값</td><td>성공</td><td>0</td></tr>
<tr><td>실패</td><td>PMFS_EPERM : dd가 읽기 모드가 아니거나, 수퍼 인덱스 채널인 경우
PMFS_EINTERNAL : PMFS 파일 시스템 내부에 문제가 발생한 경우
PMFS_EBADD : 디스크립터가 잘못 지정된 경우
PMFS_EINVALID : position이 NULL이거나 잘못된 영역인 경우,
데이터의 맨 마지막 위치 이후의 위치를 지정한 경우
PMFS_ENOINDEX : 해당 위치가 dd의 채널에 속한 데이터가 아닌 경우
PMFS_EDISKACCESS : 이전에 하드 디스크에서 에러가 난 경우</td></tr>
<tr><td>참고</td><td colspan="2">pmfs_locate_index() 시간에 의한 인덱스 채널 위치 찾아가기
pmfs_lseek() 채널에서의 논리적인 위치 이동하기
pmfs_tell() 채널의 물리적인 읽기/쓰기 위치 알아내기</td></tr>
</table>

<table>
<tr><th colspan="3">pmfs_lseek()</th></tr>
<tr><td>기능</td><td colspan="2">채널에서의 논리적인 읽기 위치를 현재 위치를 기준으로 옮긴다.</td></tr>
<tr><td>사용법</td><td colspan="2">#include <pmfsif.h>
int pmfs_lseek(int dd, int offset);</td></tr>
<tr><td>인자</td><td colspan="2">dd : pmfs_open()에 의해 얻어진 채널 디스크립터
offset : 채널에서 논리적인 읽기 위치를 이동하려는 거리</td></tr>
<tr><td>설명</td><td colspan="2">pmfs_lseek() 함수는 인덱스 및 채널의 읽기 위치를 현재 위치를 기준으로 앞, 뒤로 offset 만큼 이동하기 위한 함수이다. 쓰기 모드의 채널에서는 pmfs_lseek()가 허용되지 않는다.
이 함수는 읽기 위치를 지정하는 만큼 옮기는 것이지만, 시간 단위의 프레임으로 구성되는 인덱스 채널과 Search Log 채널의 경우 시간 경계를 넘는 곳으로 읽기 위치를 옮기는 것은 바람직하지 않다.
pmfs_lseek()는 현재 읽기 위치를 기준으로 원하는 바이트 수 만큼, 앞 뒤로 이동하는 것이 가능하지만, 가까운 거리(예를 들어, 채널에서 한 Chunk 크기보다 짧은 거리)로 읽기 위치를 이동하는 경우에만 사용한다. 통상적인 채널에서의 읽기 위치 결정은 인덱스 채널의 위치 정보를 이용하여 pmfs_iseek() 함수로 한다. (PMFS 파일 시스템에서는 인덱스 채널을 제외하고는 한 채널의 데이터 위치에 관한 전역 정보를 유지하지 않는다.)
offset이 음수일 때, offset 만큼 이동한 새로운 읽기 위치가 채널 내의 최초 데이터보다 이전이 되는 경우에는 PMFS_ENOINDEX 에러로 리턴하며 읽기 위치는 바뀌지 않는다.
Offset이 양수일 때, offset 만큼 이동한 새로운 읽기 위치가 채널 내 최후 데이터보다 이후가 되는 경우에는 채널의 맨 뒤로 읽기 위치가 지정된다.
인덱스 채널과 Search Log 채널의 경우, 데이터가 한 시간 단위로 구성되므로 그 이상의 범위로 pmfs_lseek()를 하는 것은 위험하다.</td></tr>
<tr><td rowspan="2">리턴값</td><td>성공</td><td>0</td></tr>
<tr><td>실패</td><td>PMFS_EINTERNAL : PMFS 파일 시스템 내부에 문제가 발생한 경우
PMFS_EBADD : 디스크립터가 잘못 지정된 경우
PMFS_EINVALID : offset이 Chunk 크기 보다 큰 경우
PMFS_ENOINDEX : offset이 음수일 때, 채널의 처음 위치보다 더 이전 위치로 이동하려는 경우
PMFS_EPERM : dd가 쓰기 모드 채널인 경우
PMFS_EDISKACCESS : 이전에 하드 디스크에서 에러가 난 경우</td></tr>
<tr><td>참고</td><td colspan="2">pmfs_locate_index() 시간에 의한 인덱스 채널 위치 찾아가기
pmfs_iseek() 채널에서의 물리적인 디스크 위치 찾아가기
pmfs_tell() 채널의 물리적인 읽기/쓰기 위치 알아내기</td></tr>
</table>

pmfs_seekend()		
기능	채널에서 읽기 위치를 데이터의 제일 끝으로 옮긴다.	
사용법	#include <pmfsif.h> int pmfs_seekend(int dd, struct StreamIndex *position);	
인자	dd : pmfs_open()에 의해 얻어진 채널 디스크립터 position : 채널 데이터의 끝 위치를 저장할 StreamIndex 구조에 대한 포인터	
설명	pmfs_seekend() 함수는 채널의 읽기 위치를 저장된 채널 데이터의 마지막 위치 (마지막으로 데이터를 쓴 위치 다음 바이트)로 이동하기 위한 함수이다. 인덱스 채널과 쓰기 모드의 채널에서는 pmfs_seekend()가 허용되지 않는다. position이 NULL이 아니면 이동한 결과 채널의 가장 마지막 데이터의 위치 (Chunk, Offset)이 position에 리턴된다.	
리턴값	성공	0
	실패	PMFS_EINTERNAL : PMFS 파일 시스템 내부에 문제가 발생한 경우 PMFS_EBADD : 디스크립터가 잘못 지정된 경우 PMFS_EPERM : dd가 쓰기 모드이거나 인덱스 채널인 경우 PMFS_EINVALID : position이 NULL이거나 잘못된 영역인 경우 PMFS_EDISKACCESS : 이전에 하드 디스크에서 에러가 난 경우
참고	pmfs_iseek() 채널에서의 물리적인 디스크 위치 찾아가기 pmfs_tell() 채널의 물리적인 읽기/쓰기 위치 알아내기	

<table>
<tr><th colspan="3">pmfs_tell()</th></tr>
<tr><td>기능</td><td colspan="2">채널의 현재 물리적인 읽기/쓰기 위치를 알아낸다.</td></tr>
<tr><td>사용법</td><td colspan="2">#include <pmfsif.h>
int pmfs_tell(int dd, struct StreamIndex *position);</td></tr>
<tr><td>인자</td><td colspan="2">dd : pmfs_open()에 의해 얻어진 채널 디스크립터
position : 읽기/쓰기 위치를 저장할 StreamIndex 구조에 대한 포인터</td></tr>
<tr><td>설명</td><td colspan="2">pmfs_tell() 함수는 position이 NULL이 아닌 경우, dd가 지정한 채널에서 현재 읽기/쓰기의 물리적인 위치를 position에 저장한다.</td></tr>
<tr><td rowspan="2">리턴값</td><td>성공</td><td>0</td></tr>
<tr><td>실패</td><td>PMFS_EINTERNAL : PMFS 파일 시스템 내부에 문제가 발생한 경우
PMFS_EBADD : 디스크립터가 잘못 지정된 경우
PMFS_EINVALID : position이 NULL이거나 잘못된 영역인 경우
PMFS_EDISKACCESS : 이전에 하드 디스크에서 에러가 난 경우</td></tr>
<tr><td>참고</td><td colspan="2">pmfs_iseek() 채널에서의 물리적인 디스크 위치 찾아가기
pmfs_lseek() 채널에서의 논리적인 위치 이동하기
pmfs_locate_index() 시간에 의한 인덱스 채널 위치 찾아가기</td></tr>
</table>

<table>
<tr><th colspan="3">pmfs_recycle()</th></tr>
<tr><td>기능</td><td colspan="2">PMFS 파일 시스템의 데이터를 지워 빈 공간을 마련한다.</td></tr>
<tr><td>사용법</td><td colspan="2">#include <pmfsif.h>
int pmfs_recycle(time_t time);</td></tr>
<tr><td>인자</td><td colspan="2">time : 삭제할 데이터의 최후 인덱스 시간 또는 0</td></tr>
<tr><td>설명</td><td colspan="2">pmfs_recycle() 함수는 PMFS 시스템에 저장된 데이터를 지워 빈 공간을 마련하고 새로 생긴 빈 공간의 크기를 리턴한다. 인자 time은 pmfs_locate_index() 함수에서처럼 초 단위의 시간으로, 내부적으로 이 시간과 가장 가까운 과거의 시간 단위의 시간으로 변환하여, 그 변환된 시간 이후의 인덱스 정보와 데이터만 남기고, 과거 인덱스 정보와 데이터는 모두 삭제한다. 만일 time 이 0 이면 PMFS 파일 시스템에서 가장 오래된 한 시간 분량의 데이터와 인덱스 정보를 파일 시스템에서 삭제한다.
PMFS 파일 시스템에 한 시간 이내의 데이터만 저장된 경우에는 Recycle을 하지 않고 그대로 리턴한다. PMFS 파일 시스템 전체 데이터를 지우려면 pmfs_mkfs()를 호출한다.
pmfs_recycle() 함수 내부에서는 recycle이 진행되는 동안 커널 인덱싱을 일시 정지 시킨다.
pmfs_recycle() 함수는 recycle 결과 남은 공간의 크기를 리턴한다. 이 값은 추정치로 디스크의 정확한 잔여 공간과는 약간 차이가 있는 값이다. (수퍼 블록 상의 빈 Chunk 수에 16을 곱한 뒤, 1을 뺀 값)</td></tr>
<tr><td rowspan="2">리턴값</td><td>성공</td><td>recycle 후 남은 공간의 크기 (MB 단위)</td></tr>
<tr><td>실패</td><td>PMFS_EINTERNAL : PMFS 파일 시스템 내부에 문제가 발생한 경우
PMFS_EACTIVE : 너무 최근 Data를 Recycle 하려 하는 경우
PMFS_ENOINDEX : 인덱스가 없는 과거나, 미래의 시간이 지정된 경우,
파일 시스템에 한 시간 이내의 데이터만 있는 경우
PMFS_ENODISK : HDD가 없는 경우</td></tr>
<tr><td>참고</td><td colspan="2">pmfs_write() 채널에 데이터를 쓰기
pmfs_mkfs() PMFS 파일 시스템 새로 구성하기</td></tr>
</table>

<table>
<tr><th colspan="3">pmfs_set_searchlog()</th></tr>
<tr><td>기능</td><td colspan="2">Search Log Index에 기록된 상위 48 비트의 값을 Set/Clear 한다.</td></tr>
<tr><td>사용법</td><td colspan="2">#include <pmfsif.h>
int pmfs_set_searchlog(int bit_position, int value);</td></tr>
<tr><td>인자</td><td colspan="2">bit_position : 설정할 비트 위치 (16에서 63까지)
value : 설정할 값 (0, 1)</td></tr>
<tr><td>설명</td><td colspan="2">pmfs_set_searchlog() 함수는 커널 내부의 PMFS 파일 시스템에서 주기적으로 Search Log 정보를 기록할 때 사용되는 값을 bit 별로 설정한다. 이 함수에 의해 조정된 Search Log 데이터는 커널에 의해 매 15초 마다 Search Log 채널 (PMFS_CH_SINDEX)에 기록된다.
Kernel에서 기록하는 Search Log 값은 시스템 부팅 때, 모든 비트가 0으로 초기화되며, 각 비트는 pmfsif.h에 정의된 영역 구분에 따라 다음과 같이 동작이 구분된다.
bit_position 0에서 (SI_BEGIN_LAST_STATE 1)인 위치는 15초 구간에서 PMFS 파일 시스템 내부에서 각 채널에 pmfs_write()가 호출되었는지를 확인하여 자동으로 설정한다. (이 구간은 응용 프로그램에서 값을 지정할 수 없다.)
bit_position이 SI_BEGIN_LAST_STATE에서 (SI_BEGIN_ONE_LAST 1)인 위치는 응용 프로그램이 마지막으로 설정한 값이 기록되며, 기록 후에도 값이 유지된다. 즉, 응용 프로그램이 다른 값으로 바꾸기 전에는 마지막 설정 값이 SearchLog에 계속 기록된다.
bit_position SI_BEGIN_ONE_LAST 이후의 위치는 15초 구간에서 응용 프로그램이 한번이라도 Set을 하면 SearchLog에 1이 기록되며, 기록 후 가장 마지막으로 설정한 값이 유지된다.</td></tr>
<tr><td rowspan="2">리턴값</td><td>성공</td><td>0</td></tr>
<tr><td>실패</td><td>PMFS_EINTERNAL : PMFS 파일 시스템 내부에 문제가 발생한 경우
PMFS_EINVALID : bit 위치 또는 설정할 값이 지정된 범위를 벗어난 경우</td></tr>
<tr><td>참고</td><td colspan="2">pmfs_iseek() 채널에서의 물리적인 디스크 위치 찾아가기
pmfs_locate_index() 시간에 의한 인덱스 채널 위치 찾아가기</td></tr>
</table>

pmfs_mkfs(char *dev)		
기능	하드 디스크에 새로운 pmfs 파일 시스템을 만든다.	
사용법	#include <pmfsif.h> int pmfs_mkfs(char *dev);	
인자	dev : PMFS 파일 시스템으로 초기화할 디스크 이름	
설명	pmfs_mkfs() 함수는 하드디스크 dev에 새로운 pmfs 파일 시스템을 만든다. dev는 "/dev/hda1"과 같이 하드디스크를 지정하며, NULL 인 경우 시스템의 default 하드 디스크를 사용한다. (현재 지원하지 않음) pmfs_mkfs()를 호출하기 전에 모든 디스크립터를 닫는다.	
리턴값	성공	0
	실패	PMFS_EINTERNAL : PMFS 파일 시스템 내부에 문제가 발생한 경우 PMFS_ENOMEM : 커널 내부에 버퍼가 부족하여 에러가 난 경우 PMFS_EINUSE : 현재 PMFS 파일 시스템이 사용 중임 PMFS_ENODISK : HDD가 없는 경우
참고		

<table>
<tr><th colspan="3">pmfs_stat()</th></tr>
<tr><td>기능</td><td colspan="2">현재 PMFS 파일 시스템의 상태를 읽는다.</td></tr>
<tr><td>사용법</td><td colspan="2">#include <pmfsif.h>
int pmfs_stat(struct SuperBlock *stat, int count);</td></tr>
<tr><td>인자</td><td colspan="2">stat : PMFS 파일 시스템의 상태를 읽을 위치count : stat에 할당된 버퍼 크기</td></tr>
<tr><td>설명</td><td colspan="2">pmfs_stat() 함수는 현재 PMFS 시스템의 상태를 읽는다.
실제 이 함수는 stat이 NULL이 아닌 경우, stat에 수퍼 블록 정보를 복사해준다. 수퍼 블록엔 전체 파일 시스템 크기, 사용 공간, 빈 공간, 각 채널에 관한 모든 정보가 들어 있다.</td></tr>
<tr><td rowspan="2">리턴값</td><td>성공</td><td>0</td></tr>
<tr><td>실패</td><td>PMFS_EINTERNAL : PMFS 파일 시스템 내부에 문제가 발생한 경우
PMFS_ENOSPACE : coutn가 SuperBlock 크기보다 작은 경우
PMFS_ENOPMFS : 시스템에 PMFS 파일 시스템이 없는 경우
PMFS_EINVALID : stat이 NULL이거나 잘못된 영역인 경우
PMFS_ENODISK : HDD가 없는 경우</td></tr>
<tr><td>참고</td><td colspan="2"></td></tr>
</table>

pmfs_sync()		
기능	메모리와 디스크의 정보를 완전히 일치시킨다.	
사용법	#include <pmfsif.h> int pmfs_sync(void);	
인자		
설명	pmfs_sync() 함수는 메모리에 유지되는 메타 데이터, 아직 쓰여지지 않은 버퍼의 내용들을 디스크에 써서 디스크와 메모리의 상태를 인위적으로 일치시킨다. 보통은 pmfs_d 데몬이 이 작업을 주기적으로 한다.	
리턴값	성공	0
	실패	PMFS_EINTERNAL : PMFS 파일 시스템 내부에 문제가 발생한 경우 PMFS_ENODISK : HDD가 없는 경우
참고		

<table>
<tr><th colspan="3">pmfs_set_blocking()</th></tr>
<tr><td>기능</td><td colspan="2">채널의 쓰기 모드를 Blocking 모드로 바꾼다.</td></tr>
<tr><td>사용법</td><td colspan="2">#include <pmfsif.h>
int pmfs_set_blocking(int dd);</td></tr>
<tr><td>인자</td><td colspan="2">dd : pmfs_open()에 의해 얻어진 쓰기 모드 채널 디스크립터</td></tr>
<tr><td>설명</td><td colspan="2">pmfs_set_blocking() 함수는 채널에 대한 쓰기를 Blocking 모드로 설정한다. Blocking 모드에서 채널에 쓰기를 하면, 자원 부족 등의 이유로 pmfs_write() 내부에서 블록이 될 수 있다. pmfs_open()을 쓰기 모드로 호출하면, 그 채널은 디폴트로 Non Blocking 모드로 동작한다.</td></tr>
<tr><td rowspan="2">리턴값</td><td>성공</td><td>0</td></tr>
<tr><td>실패</td><td>PMFS_EINTERNAL : PMFS 파일 시스템 내부에 문제가 발생한 경우
PMFS_EPERM : dd가 쓰기 모드가 아닌 경우
PMFS_EBADD : 디스크립터가 잘못 지정된 경우</td></tr>
<tr><td>참고</td><td colspan="2">pmfs_open() 채널 디스크립터의 할당
pmfs_write() 채널에 데이터 쓰기
pmfs_set_nonblocking() 채널의 쓰기 모드를 Non Blocking 모드로 바꾼다.</td></tr>
</table>

<table>
<tr><th colspan="3">pmfs_set_nonblocking()</th></tr>
<tr><td>기능</td><td colspan="2">채널의 쓰기 모드를 Non Blocking 모드로 바꾼다.</td></tr>
<tr><td>사용법</td><td colspan="2">#include <pmfsif.h>
int pmfs_set_nonblocking(int dd);</td></tr>
<tr><td>인자</td><td colspan="2">dd : pmfs_open()에 의해 얻어진 쓰기 모드 채널 디스크립터</td></tr>
<tr><td>설명</td><td colspan="2">pmfs_set_nonblocking() 함수는 채널에 대한 쓰기를 Non Blocking 모드로 설정한다. Non Blocking 모드에서 채널에 쓰기를 하면, 자원 부족 등의 이유로 블록을 해야 되는 순간, 블록 되지않고, PMFS_EWOULDBLOCK 에러로 리턴한다.
pmfs_open()을 쓰기 모드로 호출하면, 그 채널은 디폴트로 Non Blocking 모드로 동작한다.</td></tr>
<tr><td rowspan="2">리턴값</td><td>성공</td><td>0</td></tr>
<tr><td>실패</td><td>PMFS_EINTERNAL : PMFS 파일 시스템 내부에 문제가 발생한 경우
PMFS_EPERM : dd가 쓰기 모드가 아닌 경우
PMFS_EBADD : 디스크립터가 잘못 지정된 경우</td></tr>
<tr><td>참고</td><td colspan="2">pmfs_open() 채널 디스크립터의 할당
pmfs_write() 채널에 데이터 쓰기
pmfs_set_blocking() 채널의 쓰기 모드를 Blocking 모드로 바꾼다.</td></tr>
</table>

<table>
<tr><th colspan="3">pmfs_check_error()</th></tr>
<tr><td>기능</td><td colspan="2">가장 최근의 디스크 에러를 확인하고 에러 상태를 clear 한다.</td></tr>
<tr><td>사용법</td><td colspan="2">#include <pmfsif.h>
int pmfs_check_error(struct disk_error *error, int clear);</td></tr>
<tr><td>인자</td><td colspan="2">error : 디스크 에러의 형태를 기술한 disk_error 구조에 대한 포인터
clear : 에러 상태를 clear (1이면 에러 상태 clear, 0이면 유지)</td></tr>
<tr><td>설명</td><td colspan="2">PMFS 파일 시스템이 동작하는 동안 하드 디스크에서 물리적인 에러가 발생한 경우, 이후의 모든 pmfs_open(), pmfs_read(), pmfs_iseek(), pmfs_lseek(), pmfs_seekend(), pmfs_tell() 함수는 즉시 PMFS_EDISKACCESS 에러로 리턴한다. 디스크 에러가 나는 경우, 하드 디스크를 교체하는 것이 바람직하다.
pmfs_check_error() 함수는 커널 내부에서 발생한 에러를 확인하고, 필요하면 에러 상태를 clear 할 수 있다. 에러 상태를 clear 하면 앞에 열거된 함수들이 정상 동작을 하지만, 이전에 에러가 났던 같은 디스크 영역을 access 하는 경우 다시 에러가 발생할 가능성이 매우 높다. pmfs_read 가운데 발생한 Chunk 헤더의 에러의 경우에는 앞뒤 chunk의 내용을 이용하여 복원이 가능한 경우가 있다. 커널 내부에서 복원이 된 경우 disk_error 구조의 rebuilt 필드에 1이 기록된다. struct disk_error 의 각 필드는 다음과 같다.
int mode; 에러를 유발한 DISK 동작 PMFS_CH_READ 또는 PMFS_CH_WRITE
int start; 에러가 난 access의 block 시작 번호
int end; 에러가 난 access의 block 끝 번호
int access_type; 에러를 유발한 access 분류
int rebuilt; Chunk Header에 에러가 났었지만 성공적으로 복원된 경우 1, 아니면 0.
access_type은 HDD_BLOCK_REQ, HDD_BUFFER_REQ, HDD_CHUNK_REQ, HDD_SUPER_REQ가운데 하나로 각각 단일 블록, 데이터영역, Chunk 헤더, 수퍼 블록 acces를 의미한다.</td></tr>
<tr><td rowspan="2">리턴값</td><td>성공</td><td>0 (error에 커널 내의 에러 상황을 리턴)</td></tr>
<tr><td>실패</td><td>PMFS_EINTERNAL : PMFS 파일 시스템 내부에 문제가 발생한 경우
PMFS_EINVALID : err 가 잘못된 메모리 영역을 가리키는 경우</td></tr>
<tr><td>참고</td><td colspan="2">pmfs_open() : 채널 읽기/쓰기를 위한 디스크립터 할당
pmfs_read() : 채널에서 데이터 읽기
pmfs_iseek() : 채널에서의 물리적인 디스크 위치 찾아가기
pmfs_lseek() : 채널의 읽기/쓰기 위치 이동
pmfs_seekend() : 채널 데이터 끝으로 읽기 위치 이동
pmfs_tell() : 채널의 현재 물리적인 읽기/쓰기 위치를 알아내기</td></tr>
</table>

[부 록 D] PMFS의 커널 내 제어 함수들

다음 함수들은 PMFS 파일 시스템 내부 (즉 라이브러리 내부), 파일 시스템 검사 (fsck) 과정에서 사용되는 함수들로 응용 프로그램에서의 직접적인 사용이 권장되지 않는다.

int pmfs_lock(void);
int pmfs_unlock(void);
파일 시스템으로 들어가는 모든 입출력을 일시 정지 시킨다. 이 함수는 recycle, file system check 등 다른 입출력 동작이 진행되는 경우, 심각한 동기화 문제가 발생할 가능성이 있는 함수가 커널 내부에 진입하는 것을 일시적으로 통제하고 해제하는 함수들이다.

int _pmfs_lock_and_sync(void);
이 함수는 lock과 sync를 연이어 실행하는 함수이다.

int _pmfs_get_block(void *buffer, int block);
int _pmfs_put_block(void *buffer, int block);
하드 디스크의 물리적인 한 블록을 읽거나 쓰는 함수이다. 이 두 함수는 파일 시스템 검사 때, 하드 디스크 블록을 직접 access 하기 위해서 사용한다.

int _pmfs_detach_chunks(void);
이 함수는 파일 시스테 검사 때, 앞서 실행된 응용 프로그램이 커널 내부에 Chunk 헤더, 버퍼 등의 정보를 유지하고 있는 경우, 그 모든 자료 구조를 flush 하고 free 하도록 하는 함수이다.

int _pmfs_reload_sb(void);
파일 시스템 검사 후, 수퍼 블록이 바뀐 경우 새로 수퍼 블록을 읽어 들이기 위한 함수 이다.

int _pmfs_num_user(void);
현재 PMFS 파일 시스템을 이용하기 위해서 열린 모든 descriptor의

개수를 리턴한다. 인덱싱이 시작되면 응용 프로그램들이 pmfs_open()으로 얻은 descriptor 개수에 수퍼 인덱스, 인덱스, Search Log 등 3이 추가된 값이 리턴된다.

char *_pmfs_err_str(int errno);
에러 번호에 해당되는 에러 스트링을 리턴한다.

int _pmfs_dump_stat(int what);
PMFS 파일 시스템 내부의 상태를 출력한다. (cat /proc/pmfs/* 와 동일)
What 이 0이면 /proc/pmfs/channel
1이면 /proc/pmfs/user
2이면 /proc/pmfs/stat
3이면 /proc/pmfs/perf
4이면 /proc/pmfs/disk
이 함수는 kernel 내부에서 printk를 사용하므로 응용 프로그램의 메시지와 엉겨서 출력될 수 있다.

[부 록 E] PMFS의 API 이용 예제

* 통상적인 데이터 쓰기

```
#include <pmfsif.h>

int dd_s, count, err;
char buffer[MY_BUF_SIZE];

dd_s = pmfs_open(PMFS_CH_1, PMFS_CH_WRITE);
while (count = get_frame_data(buffer)) {
     if ((err = pmfs_write(dd_s, buffer, count)) != count) {
        /* 에러 check */
        switch (err) {
           case PMFS_ENEEDRECYCLE :
                { Recycle }
                break;
           case PMFS_EWOULDBLOCK :
                { Do something }
                break;
           default :
                { CALL AS }
                break;
        }
     }
}
pmfs_close(dd_s);
```

*. 원하는 시간, 원하는 채널에서 데이터 읽기

```
#include <time.h>
#include <pmfsif.h>

int dd_s, dd_i;
time_t time;
i64 offset;
struct StreamIndex where;
char buf[MY_BUF_SIZE];

dd_i = pmfs_open(PMFS_CH_INDEX, PMFS_CH_READ); /* 채널을 연다 */
{ 15초 단위의 time을 만든다 }
pmfs_locate_index(dd_i, time);
        /* 필요하면 offset을 지정하여 pmfs_lseek()를 한다. */
pmfs_lseek(dd_i, offset);
         /* 해당 인덱스 정보가 있는 곳까지 이동 */
pmfs_read(dd_i, &where, sizeof(where));
         /* where에 offset에 해당되는 데이터의 위치를 읽어온다 */
pmfs_close(dd_i);
if (!where.chunk)
        { ERROR ,   NO DATA ! ,   RETURN}
        /* 그 시간에 데이터를 위한 인덱스가 없음 */

dd_s = pmfs_open(PMFS_CH_2, PMFS_CH_READ);        /* 해당 채널을 연다
*/
pmfs_iseek(dd_s, &where);                  /* 원하는 데이터의 위치로 간다 */

while (stop_command) {
        pmfs_read(dd_s, buf, MY_BUF_SIZE);
        play_frame(buf);
        if (forward) {
                ...
        } else {
                ...
```

*. Disk Error가 발생한 경우, 처리

디스크 Error가 발생한 것으로 리턴한 경우, 만일 마지막 에러가 Chunk Header를 읽다가 발생한 경우에는 PMFS 파일 시스템 내부에서 복원이 가능한 경우가 있다. 그 경우, 원하면 다음과 같이 프로그램을 중단하지 않고 계속 처리가 가능하다.

```
if ((count = pmfs_read(dd, Buf, readsize)) <= 0)  {
      if (count == PMFS_EDISKACCESS) {
        struct disk_error error;
        pmfs_check_error(&error, 1);
        if ((error.mode == PMFS_CH_READ) &&
            (error.access_type == HDD_CHUNK_REQ) &&
              /* Chunk 헤더에 대한 Read */
            error.rebuilt)
                        /* 성공적으로 복원됨 */
                       // 계속 read operation이 가능
        else
                // 복원 되지 않은 에러
      }
      break;
}
```

디스크 Error가 인덱스 정보를 포함한 데이터 영역에서 발생한 경우에는 에러가 난 블록에 Align된 32KB 영역 (PMFS 파일 시스템 내부의 버퍼 유지 단위) 전체를 읽지 않은 것처럼 동작하며, pmfs_disk_error()로 clear 된 뒤 다시 읽을 수 있다. (일단 에러가 난 디스크 영역은 다시 에러가 나기 쉽기 때문에 같은 일이 발생할 가능성이 높다.)

[부 록 F] 유틸리티 명령들 (TBD)

mkfs

hdd pmfs 파일 시스템을 만든다.
모든 데이터를 지울 때도 사용
(현재는 hdd-device 인자는 사용하지 않음)

fsck

PMFS 파일 시스템을 검사한다.
(현재는 hdd-device 인자는 사용하지 않음)

stat

PMFS 파일 시스템의 Super Block 정보를 출력한다.

cat /proc/fs/pmfs

PMFS 파일 시스템 드라이버의 현재 상태를 보여 준다.

ABSTRACT

The Implementation of Linux File System for Personal Multimedia System

Sohn, Jungsoo
Computer Engineering
of the Graduated School
Hansung University

In Personal Multimedia Systems, very large files are stored and played. Because those files are read-centric and have low reusability, legacy Linux file systems that are optimized for the use of many small files, do not show good performance in the Personal multimedia systems. I implement a new file system, PMFS, to deal with those characteristics. The new file system shows the superior performance in read, write and mixed operations to the Linux's EXT2 or EXT3 file systems. At the same time, PMFS has much better stability to the unexpected power-failure. The new file system is implemented on the Linux operating system, and it also can be easily ported to other operating systems.

감사의 글

지난 대학원 생활을 돌아볼 때 크고 작은 많은 경험을 했음을 깨닫게 됩니다. 그리고 제가 감사드려야 할 분이 너무도 많이 계시다는 사실도 알게 됩니다. 그러나 이 시간 가장 먼저 저를 한성대 대학원으로 인도하시고 최고의 교수님을 만나게 해주시고 또한 모든 어려움 가운데서 저를 지켜주신 살아계신 하나님 아버지께 감사와 찬양을 돌려 드립니다. 처음 대학원에 들어와 컴퓨터를 전공하며 학업에 대한 무지로 고생을 많이 했습니다. 그러나 하나님은 저의 목자 되셔서 그 선하신 섭리와 손길로 저를 인도하여 주셨습니다.

조에스라 목자님 가정, 양희직 목자님 가정, 또한 그 밖의 한성대 동역자들의 기도와 사랑에 감사드립니다. 희직 목자님의 도움으로 대학원에 들어올 수 있었고 에스라 목자님을 통해 대학원 생활에서 실제적인 투쟁을 할 수 있었습니다.

지도 교수이신 이민석 교수님께 감사를 드립니다. 교수님은 시킨 일도 잘 못하는, 학생으로서 자질이 부족한 저를 인내해 주셨고 전공에 대한 많은 지식을 가르쳐 주셨으며 또 프로젝트를 통해 등록금 문제를 도와 주셨습니다. 또한, 이 논문을 완성하도록 실제적으로 도와 주셨습니다. 교수님께 다시 한번 감사드립니다. 바쁘신 가운데에서도 저의 논문을 심사해 주신 김영웅 교수님, 황기태 교수님, 정인환 교수님께 감사를 드립니다.

마지막으로 고향에서 밤낮 고생하시며 저를 위해 기도해 주시는 아버지와 어머니의 깊은 사랑에 감사를 드립니다. 대학에 들어와서부터 줄 곧 바쁘다는 핑계로 거의 집에 내려가지 못했지만 저를 믿어주시고 기도해 주신 부모님의 은혜는 결코 잊을 수 없습니다. 이 부모님의 은혜에 다시 한번 감사를 드립니다.

이 논문은 보기에도 작은 것입니다. 하지만 이것이 컴퓨터 분야에 조금이라도 도움이 되기를 기원합니다. 앞으로 계속해서 전공분야에 매진하여 전문가로서 성장하고자 합니다. 저의 이 작은 비젼을 통해서 하나님께서 영광을 받으실 것을 믿습니다.

손 정 수