논문 2006-01-13

멀티미디어 데이터 스트림을 위한 파일 시스템의 설계 및 구현

(A New File System for Multimedia Data Stream)

이 민 석*, 송 진 석
(Minsuk Lee, Jin-Seok Song)

Abstract : There are many file systems in various operating systems. Those are usually designed for server environments, where the common cases are usually 'multiple active users', 'great many small files' And they assume a big main memory to be used as buffer cache. So the existing file systems are not suitable for resource hungry embedded systems that process multimedia data streams. In this study, we designed and implemented a new file system which efficiently stores and retrieves multimedia data steams. The proposed file system has a very simple disk layout, which guarantees a quick disk initialization and file system recovery. And we introduced a new indexing-scheme, called the time-based indexing scheme, with the file system. With the indexing scheme, the file system maintains the relation between time and the location for all the multimedia streams. The scheme is useful in searching and playing the compressed multimedia streams by locating exact frame position with given time, resulting in reduction of CPU processing and power consumption. The proposed file system and its APIs utilizing the time-based indexing schemes were implemented firstly on a Linux environment, though it is operating system independent. In the performance evaluation on a real DVR system, which measured the execution time of multi-threaded reading and writing, we found the proposed file system is maximum 38.7% faster than EXT2 file system.

Keywords : file system, time-based indexing, multi-media stream, embedded system

Ⅰ. 서 론

초고속 네트워크를 기반으로 하는 멀티미디어 서비스가 일반화되고, TV 등 기존의 영상, 음향 가전이 고기능, 디지털화되면서, 대용량의 멀티미디어 데이터 스트림을 저장, 재생하는 기능이 많은 임베디드 시스템에서 요구되고 있다.

특히 영상 및 음향을 실시간으로 저장, 재생하는 보안 장치인 DVR (Digital Video Recorder), 방송되고 있는 내용을 저장하면서 동시에 재생을 함으로써 방송 중인 화면을 정지하거나, 지난 화면으로 되돌아가서 재생하는 기능을 가진 PVR (Personal Video Recorder), 하드 디스크에 음악 파일을 저장하는 MP3 기기, 기존의 자기 테이프 대신 대용량 플래쉬 메모리, 기록 가능 DVD 매체, 하드 디스크에 영상을 저장하는 캠코더 등은 저장 매체의 디지털화에 따른 여러 가지 이점 때문에 수 년 전부터 시장에 도입되기 시작했으며, 현재는 기존 아날로그 기반의 시장을 빠르게 대치해가고 있다.

기존에 존재하던 많은 파일 시스템들은 주로, 많은 동시 사용자, 작은 파일 위주의 참조라는 서버 환경을 가정하여 설계되어, 대용량의 멀티미디어 데이터를 효율적으로 저장하고 재생하기에는 적합하지 않은 경우가 많으며, 특히 최근에 만들어진 서버용 파일 시스템들의 경우는 위에 언급한 서버 환경과 멀티미디어 환경을 모두 고려하여 설계되어 서버 시스템에서는 다양한 데이터 참조에 좋은 성능을 발휘하고 있다. 하지만 이와 같은 서버 위주의 파일 시스템들은 이를 구현한 소프트웨어의 크기도 크고, 컴파일 된 실행 바이너리 크기도 크며, 파일 시스템의 성능을 메타 데이터 및 파

* 교신저자
논문접수 : 2006. 06. 08. 채택확정 : 2006. 09. 07.
이 민석, 송 진석 : 한성대학교 컴퓨터공학과
※ 본 논문은 산학협동재단의 2005년 학술연구비 지원에 의한 연구 결과임.

일 데이터에 대한 버퍼링에 의존하기 때문에 대용량의 메인 메모리를 가진 시스템에서만 효과적인 경우가 많다. 따라서 CPU 성능, 메인 메모리의 용량, 프로그램을 저장하기 위한 플래쉬 메모리 공간 등 여러 가지 자원에 제약이 많은 임베디드 시스템 환경에서 기존 파일 시스템을 멀티미디어 데이터 스트림 저장을 위한 도구로 사용하는 것은 상당한 무리가 있다.

이에 본 연구에서는 임베디드 시스템 환경에 적합한 새로운 멀티미디어 파일 시스템을 설계 구현하였다. 구현된 파일 시스템은 멀티미디어 스트림을 저장하는 여러 장치들이 요구하는 여러 가지 기능 및 성능 요구 사항들을 만족하면서, 운영체제 종속적 요소들을 배제하여 많은 임베디드 시스템에 적용될 수 있도록 설계되었으며, 멀티미디어 스트림을 다루는 응용 프로그램들에 필요한 다양한 기능을 쉽게 구현할 수 있는 새로운 파일 시스템 API를 기존 파일 시스템 접근 방법에 추가하여 제공하고 있다.

이 새로운 파일 시스템은 연속적인 멀티미디어 데이터를 생성, 저장하고, 검색, 재생하는 모든 멀티미디어 기기에 적용 가능하다. 대표적인 목표 시스템으로는 DVR, PVR을 들 수 있다. 두 시스템은 연속적인 영상 데이터를 저장하고, 실시간을 인자로 하는 검색과 재생을 하는 시스템이다. 두 시스템에서, 제안된 파일 시스템은 영상 데이터와 인덱싱 정보의 저장, 가장 최근 데이터를 저장하기 위한 오래된 데이터 영역의 리사이클, 주어진 시간에 해당하는 프레임 위치 검색 기능을 지원하며, 순차 읽기에서의 높은 성능을 제공한다.

본 논문의 구성은 다음과 같다. 제 II 장에서는 연구 배경으로, 기존 파일 시스템들에 대한 간단한 리뷰를 하고, 새로운 파일 시스템의 필요성과 요구 사항을 정리한다. 제 III 장에서는 이 연구에서 설계된 파일 시스템과 설계된 파일 시스템을 이용하기 위한 API, 그리고 제안된 파일 시스템이 가진 제약과 극복 방안에 대하여 구체적으로 기술한다. 제 IV 장에서는 설계된 파일 시스템의 성능을 평가하기 위하여 EXT2 파일 시스템과 대용량 멀티미디어 파일에 대한 초기화, 읽기/쓰기에 대한 성능을 비교, 분석한다. 마지막으로 제 V 장에서는 논문을 요약한다.

II. 연구 배경

1. 기존 파일 시스템

파일 시스템은 보통 운영체제의 일부로서 주로 컴퓨터의 하드 디스크에 데이터를 저장하고, 읽어내는 논리적인 수단이다. 즉, 파일 시스템은 파일과 디렉터리에 관한 모든 정보를 의미하는 메타데이터의 자료 구조와 파일 내의 실제 데이터가 디스크에 저장되는 방식을 정하여 관리하며, 응용 프로그램이 파일을 읽고, 쓰는 연산, 데이터의 버퍼링, 초기화 및 오류에서의 복구 등 파일에 관련된 모든 동작을 관리한다.

현재까지 개발된 파일 시스템은 수도 없이 많다. 우선 마이크로소프트의 윈도우 NT 계열 운영체제에서 사용하는 NTFS[1]와 이전의 MS-DOS 환경과 USB 디스크와 같은 이동식 매체에서 사용하는 VFAT 파일 시스템[2], CD나 DVD와 같은 광매체에서 사용되는 ISO-9660[3], UDF[4]와 같은 파일 시스템 등이 있다. 이 논문에서는 공개 소스 형태로 개발되고 많은 임베디드 시스템에 적용되는 리눅스 운영체제에서 접근 가능한 EXT2[5]와 저널링 파일 시스템들인 EXT3[6], JFS[7], XFS[8], ReiserFS[9] 등 몇 가지 파일 시스템에 대하여 간략히 소개하고자 한다.

EXT2(The Second Extended File System)는 Way Davidson에 의해 설계된 파일 시스템으로써 페이지 단위가 아닌 Extent 라는 단위로 블록을 관리하며, 자유 공간이나 사용 블록 관리가 쉽다. EXT2 파일 시스템은 작은 크기의 파일이 많으면 페이지 단위 관리를 하는 다른 파일 시스템들보다 좋지 않은 성능을 보인다. EXT3(The Third Extended File System)는 EXT2에 저널링 기능을 도입한 EXT2의 확장 파일 시스템이다. EXT3를 포함한 저널링 파일 시스템들은 최근의 디스크 쓰기 연산만을 가지고 있는 특별한 영역인 저널을 살펴봄으로써, 오랜 시간이 필요한 전체 파일 시스템에 대한 일관성 검사를 피할 수 있다. 현재 EXT3는 리눅스의 표준 파일 시스템으로서 많은 서버, 워크스테이션에서 널리 사용되고 있다. ReiserFS는 Namesys의 Hans Reiser와 그가 이끄는 팀이 개발한 파일 시스템으로 작은 파일에 대한 참조 성능 향상에 초점을 맞추어 설계되었으며, 디렉터리, 파일, 데이터를 구성하는 방법으로는 B* 트리를 사용한다. JFS(Journaled File System)는 IBM에서 만들어진 파일 시스템으로 현재 IBM

의 엔터프라이즈 서버에서 사용되고 있다. JFS는 서버 환경에서의 높은 처리량을 위해 설계된 파일 시스템으로서 메타 데이터 저널링을 지원하며 메타 데이터를 B+ 트리로 관리한다. XFS는 블록 크기를 512B에서 64KB까지 지정할 수 있으며, 메타 데이터 저널링을 지원한다. 또 메타 데이터 관리를 위해 B* 트리를 사용하며 완전한 64비트 파일 시스템으로 수백만 TB의 용량을 지원한다.

그 밖에도 멀티미디어 시스템을 위한 전용 파일 시스템에 관한 여러 연구가 진행되어 왔다. 대부분의 연구는 서버에서 멀티미디어 파일을 지원하기 위한 연구들로 임베디드 시스템을 지향하고 있는 본 연구와는 차별된다. 즉, 파일 시스템의 구조보다는 멀티미디어 데이터의 시간 동기성 지원을 위한 운영 체제의 역할, 멀티미디어 파일 재생을 효과적으로 지원하기 위한 선인출 기법 등에 집중되고 있다. 우선 Niranjan 등의 MMFS 연구[10]에서는 VFS 구조 위에서 범용 파일 시스템과 멀티미디어 데이터 처리를 같이 할 수 있는 파일 시스템을 개발하였다. 이 연구는 향상된 미리 읽기와 선인출, 그리고 우선순위에 따른 디스크 스케줄에 치중하고 있다. 하지만 이 연구에서의 선인출 정책은 가변 비트율 멀티미디어 데이터의 프레임 크기를 반영하지 못함으로써 추가적인 CPU 연산을 필요로 한다는 한계를 가진다. 또 IBM의 Tiger Shark [11] 파일 시스템도 큰 디스크 블록, 확장된 API, 자원 예약과 진입 제어 등을 지원하나, 프레임 동기화 등을 지원하지 못하고 있다. 또 분산형 멀티미디어 저장 시스템인 Fellini[12]와 MARS 프로젝트의 일환으로 제안된 방법[13]들은 분산된 멀티미디어 데이터 참조에 대한 실시간 응답을 목표로 하는 VOD 서버 시스템을 대상으로 하는 연구로 임베디드 환경에 적합하지 않다. 그 외에도 멀티미디어 스트림을 위한 디스크 스케줄링 알고리즘에 관한 연구[14][15], 데이터 형태와 무관한 파일 시스템과 그 위에서 데이터 타입과 직접 연관된 정책들을 수행하는 계층적 파일 시스템 연구[16] 등 많은 연구가 있었으며, 특별한 하드웨어의 지원으로 주문형 스트리밍 서비스를 가속하며, 이를 기반으로 EXT3 파일 시스템을 확장한 연구[17]도 진행된 바 있다.

이전에 진행된 거의 모든 멀티미디어 파일 시스템 연구는 서버 수준의 시스템, 분산된 VOD 서버를 위한 파일 저장 장치를 위한 방법들에 관한 연구이며, 많은 경우 운영체제에서 여러 멀티미디어 스트림을 동시에 참조할 때 요구되는 시간 제약성의 만족을 위한 프로세스, 네트워크 자원, 디스크 참조 우선순위 등의 제어에 집중하고 있다. 반면에 본 논문에서는 독립적인 임베디드 시스템으로서 멀티미디어 스트림 처리를 하는 경우, 참조 성능과 초기화 및 복구 시간 단축을 위한 디스크 배치 구조, 응용 프로그램이 멀티미디어 스트림에 접근할 때 원하는 프레임에 적은 비용으로 접근하도록 하는 인덱싱 구조를 제안함으로써 이전 연구와 차별되며, 멀티미디어 파일 시스템에 관한 이전 연구 결과는 부분적으로 본 연구에서 개발된 파일 시스템에도 적용할 수 있을 것으로 사료된다.

2. 멀티미디어 스트림 저장을 위한 파일 시스템의 필요성과 요구 사항

멀티미디어 데이터란 일반적으로 텍스트, 이미지, 그래픽, 비디오, 오디오 등 여러 가지의 미디어 데이터 정보가 하나로 융합되어 디지털 방식으로 표현된 데이터를 의미하며, 주로 파일이나 스트림 형태로 접근한다. 멀티미디어 스트림은 멀티미디어 데이터의 지속적인 흐름을 의미한다.

멀티미디어 파일 및 스트림을 저장하기 위한 멀티미디어 파일 시스템은 이 파일 시스템이 사용되는 환경에서 제공되는 멀티미디어 서비스와 사용자 관점에서 보는 멀티미디어 기기의 품질을 유지하기 위하여 다음과 같은 요구 사항들을 가지고 있다.

첫째, 주로 비디오, 오디오 등의 정보를 저장하므로, 파일 또는 단위 데이터의 크기가 적어도 수 MB에서 수십 GB에 이르기까지 매우 큰 용량을 가지고 있다. 따라서 큰 용량의 데이터 참조에 효과적인 파일 시스템 구조가 필요하다.

둘째, DVR, PVR과 같은 멀티미디어 기기에서는 파일이라는 개념보다는 채널 단위의 스트림 데이터가 전체 디스크를 사용하고 있다. 즉, 단위 데이터인 파일이 아닌 연속적인 스트림을 저장, 재생, 삭제하기 위한 파일 시스템 구조와 스트림 참조를 위한 저장 구조와 API 가 필요하다.

셋째, 멀티미디어 데이터에 대한 접근에 있어서 저장과 재생은 대부분 순차적으로 수행된다. 따라서 데이터의 저장과 저장된 데이터에 대한 순차 참조 성능이 극대화될 수 있도록 파일 시스템이 구성되어야 한다. 이는 파일 시스템에서 디스크에

데이터를 저장할 때와 저장된 데이터를 읽을 때, 데이터가 가지고 있는 순차성을 이용하여야 한다는 것으로 데이터 블록의 크기를 크게 하고, 적극적인 미리 읽기 등에 대한 고려를 해야 함을 의미한다.

넷째, 멀티미디어 데이터는 원초적으로 시간 동기성을 가진다. 따라서 멀티미디어 파일은 저장 및 재생에 있어서 데이터의 바이트 위치가 아닌 시간을 기준으로 하는 인덱싱 기법이 필요하다. 또한 시간 기준 인덱싱은 기존 파일 시스템에 유지하는 위치 기준 인덱싱 기법과 근본적으로 다르기 때문에 이를 활용할 수 있는 새로운 API도 정의되어야 한다.

본 연구에서는 위의 멀티미디어 데이터와 관련된 요구 사항과 임베디드 시스템을 위한 요구 사항들, 즉 파일 시스템 자체를 구현한 소프트웨어의 크기를 줄이고, 파일 시스템 초기화 시간 단축, 오류에 대한 파일 시스템의 안정성 (복구되어 운영될 가능성과, 복구 시 데이터 손실의 최소화) 제고, 복구 시간의 단축, 여러 운영체제에의 이식 가능성 등을 만족하는 새로운 파일 시스템을 설계하고 구현하였다. 또 파일 시스템이 제공하고 있는 시간 기준의 인덱싱, 스트림 저장을 위한 특화된 기능들을 응용 프로그램이 쉽게 이용할 수 있도록 하는 새로운 API들을 기존의 파일 시스템 접근을 위한 POSIX API에 추가하여 제공하고 있다.

Ⅲ. 멀티미디어 스트림 저장을 위한 파일 시스템

본 연구에서 제안한 멀티미디어 스트림 저장을 위한 파일 시스템(이하 MFS, Multimedia File System)은 대용량 멀티미디어 스트림을 저장하고, 재생하는 임베디드 응용에 적합하도록 다음과 같은 목표들이 고려되어 설계되고 개발되었다.

- 멀티미디어 데이터에 대한 순차 참조 최적화
- 시간 기준 인덱스에 의한 탐색
- 파일 시스템 초기화 시간 단축
- 파일 시스템 검사 및 복구 시간 최소화
- 파일 시스템 복구 가능성 및 데이터 복구율 최대화
- 다양한 운영체제 환경에의 이식 가능성 확보

위와 같은 설계 목표를 달성하기 위하여 연구에서는 파일 시스템의 구조를 단순화하고 유닉스 계열 운영체제들이 사용하는 VFS(Virtual File System)를 설계에서 배제하여 임베디드 시스템들이 사용하는 다양한 운영체제에서 동작할 수 있도록 하였으며, 새로운 파일 시스템 접근 API를 개발하여 응용 프로그램들이 멀티미디어 스트림 데이터에 효율적으로 접근할 수 있도록 하였다.

1. 파일 시스템의 레이아웃

MFS는 그림 1과 같이 매우 단순한 데이터 배치 구조를 갖는다. 이러한 단순한 데이터 배치 구조는 특히 새 디스크를 파일 시스템으로 초기화할 때, 디스크에 기록해야 하는 데이터의 양의 최소화함으로써 초기화 시간을 줄인다. 또 불시의 전원 차단 후 다시 시스템이 켜졌을 때 실시하는 파일 시스템 복구 때도, 무결성 검사의 대상이 되는 자료 구조가 간단하여 복구 시간을 줄일 수 있고, 데이터의 복구 가능성도 높이는 역할을 한다. 이러한 기능은 수 백 GB에 이르는 대용량 하드 디스크를 사용하는 경우, 무정지 전원에 의한 백업이 가정되지 않는 가전형 임베디드 시스템 형태의 멀티미디어 기기에서 상당히 유용하다.

실제 초기화 과정에서는 하드 디스크의 중앙에 메타 데이터와 복구를 위한 백업 메타 데이터를 배치하고, 남은 영역을 큰 크기의 데이터 블록으로 채우고 있다.

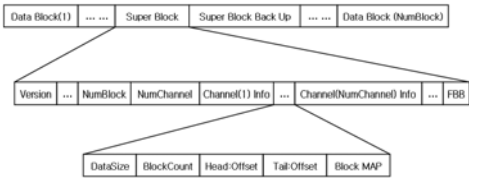

그림 1. MFS의 디스크 배치
Fig. 1. Disk Layout of MFS

MFS는 단순한 디스크 데이터 배치를 가지고 최대 채널수와 단위 데이터 블록의 크기만을 지정하여 초기화를 함으로써, 현존하는 수 백 GB 용량의 하드디스크에 대해서도 순간적으로 초기화를 완료할 수 있으며, 불시의 전원 차단 후에도 수 초 이내에 파일 시스템 검사와 복구가 가능하다.

표 1. 주요 메타 데이터
Table 1. Key Meta Data

BlockSize		단위 데이터 블록크기
NumBlock		전체 데이터 블록 수
NumFreeBlock		빈 데이터 블록 수
NumChannel		데이터 채널 수
채널별	*DataSize*	총 바이트 수
	BlockCount	데이터 블록 수
	Head	시작 위치
	HeadOffset	맵에서의 시작 블록의 오프셋
	Tail	마지막 위치
	TailOffset	맵에서의 마지막 블록의 오프셋
	Map	데이터 블록 할당 맵
IndexInterval		인덱싱주기
FirstIndexTime		데이터가 최초로 기록된 시간
LastIndexTime		마지막으로 데이터가 기록된 시간
FBB		할당되지 않은 데이터 블록 비트맵

MFS의 주요 메타 데이터는 표 1과 같다. 메타 데이터 가운데 BlockSize 와 NumChannel은 하드 디스크에 파일 시스템을 초기화할 때 지정되는 값이며, FirstIndexTime과 LastIndexTime은 파일 시스템이 DVR와 같은 기기에서 사용될 때 채널별, 시간 별 로 유지되는 인덱스의 시작과 끝 시간에 대한 정보이다.

각 채널 정보에는 이 채널에 할당된 데이터 블록들의 번호가 차례로 저장되는 데이터 블록 할당 맵이 있다. 전체 디스크 크기를 수용하는 이 블록 할당 맵에 의하여 MFS는 그림 2와 같이 첫 데이터 블록 번호가 저장된 맵 엔트리, 그 데이터 블록에서의 시작 오프셋, 마지막 데이터 블록 번호가 저장된 엔트리와 그 블록에서의 마지막 오프셋을 유지하는 간단한 데이터 블록 맵핑 방법을 유지하여 운영된다.

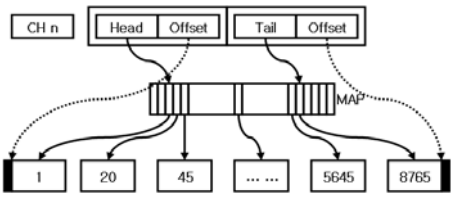

그림 2. 각 채널의 데이터 블록 지정 방법
Fig 2. Data Block Mapping for a Channel

2. 시간 기준 인덱싱 기법

앞서 설계 목표에서 언급한 바와 같이, 임베디드 멀티미디어 기기에서는 파일 이름이나 절대적인 바이트 위치에 의한 방법이 아니라, 각 채널에서 시간을 기준으로 데이터의 위치를 찾는 방법이 필요하다.

시간 기준 인덱싱 기법은 특히 압축된 멀티미디어 파일의 임의 재생에서 매우 유용하다. 일반적으로 압축 비율을 높이기 위하여 멀티미디어 파일은 가변 비트율로 압축된다. 이는 영상의 경우 각 프레임, 또는 프레임 조합의 크기가 화면 내용에 따라 다르다는 것을 의미한다. 압축된 멀티미디어 스트림 데이터를 재생하는 도중 사용자가 FF (fast forward), REW (rewind) 버튼을 누르거나, 10배속 또는 100배속 등의 고속 재생을 요구할 때, 기존 파일 시스템을 이용하는 경우에는 응용 프로그램이 평균 프레임 크기를 고려하여 적당한 파일 위치로 이동한 뒤에 데이터 스트림을 읽어가면서 분석하여 새로운 프레임, 또는 재생 가능한 프레임 조합의 시작 위치를 찾는 작업을 하게 된다. 이 작업으로 당연히 CPU의 작업 부하가 증가하여 응답이 늦어지고, 전력 소모도 증가하게 된다.

하지만 시간 기준 인덱싱이 있는 경우, 지정한 시간에 해당하는 프레임 시작 위치를 인덱스 정보를 이용하여 바로 찾을 수 있어, 불필요한 연산을 없앨 수 있다.

인덱싱 주기는 응용 프로그램의 요구에 따라 임의의 시간으로 결정될 수 있다. 예를 들어 MPEG에서는 한 GOP(Group of Pictures)가 보통 5에서 60개의 프레임으로 구성된다[18][19]. 이는 GOP가 165msec에서 1980msec의 시간에 해당하는 프레임으로 구성된다는 것으로 MFS 파일 시스템에서는 인덱싱 주기를 1/framerate로 지정하여 세부적인 프레임 단위로 스트림의 위치를 직접 찾아갈 수 있도록 하거나, 인덱싱 위치가 항상 GOP 경계 위치가 되도록 할 수 있다. 이 방법으로 스트림을 읽어가면서 프레임 또는 프레임 조합의 시작 위치를 찾기 위한 CPU 자원, 전력 낭비를 줄일 수 있다.

MFS에서는 시간 기준 인덱싱을 지원하기 위해서 인덱스 정보만을 저장하는 인덱스 채널을 따로 정의하였으며, 이 인덱스 채널에는 지정된 인덱싱 주기마다 각 채널의 데이터가 어떤 데이터 블록의 어느 위치에 있는지를 모든 채널에 대하여 기록한다. 그림 3은 MFS가 유지하는 인덱싱 구조를 나타낸다. 그림 3의 제일 위 부분은 첫 인덱싱 시간부터 마지막 인덱싱 시간까지의 가상적인 시간을 의미하며, 각 인덱싱 주기마다 각 채널에서의 데이

터 시작 위치를 모두 기록하는 것을 보여주고 있다. 그림에서 채널 1을 예로 들면, 첫 인덱싱 때 데이터 블록 833의 첫 위치에 데이터를 쓰기 시작한 채널 1은 두 번째 인덱싱 주기가 시작될 때, 두 블록의 데이터를 채우고 데이터 블록 324번의 23,458 바이트 위치를 쓰고 있다는 것을 알 수 있다. 즉, 시간 기준 인덱싱 기법을 이용하면 주어진 시간을 인자로 그 시간에 해당하는 스트림 데이터의 위치를 정확하게 알아낼 수 있다.

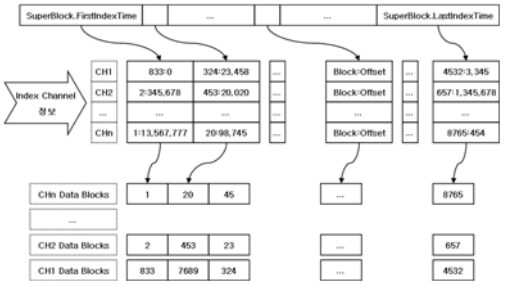

그림 3. MFS의 인덱싱 정보

Fig 3. Indexing Information of MFS

3. 멀티미디어 파일 시스템을 위한 API

MFS의 설계에서는 유닉스 계열 운영체제들이 사용하는 VFS를 배제하여, 실시간 운영체제와 같이 임베디드 시스템들이 사용하는 다양한 운영체제, 또는 운영체제가 없는 환경에도 쉽게 이식될 수 있도록 하였다. 이는 VFS가 파일 시스템과 시스템 자원에 접근하기 위하여 제공하는 기본적인 구현 틀과 자료 구조들, 그리고 버퍼 캐쉬 기반을 MFS에서는 사용하지 않고 모든 자료 구조 및 버퍼 관리를 직접 한다는 것을 의미한다.

또한 VFS를 이용하지 않기 때문에, MFS는 시간 기준 인덱싱 관련 기능과 디스크 공간 재활용 등 새로운 기능 등은 물론, 기본적인 파일 접근 API를 지원하기 위하여 파일 시스템을 제어하는 디바이스 드라이버 형태의 모듈을 만들어, 디바이스 드라이버의 ioctl() API의 개별적인 명령 형태로 다양한 응용 프로그램에 대한 서비스를 수행한다. 하지만 각 명령은 사용자 라이브러리 수준에서 기존 POSIX API 형태로 추상화되어, 응용 프로그램 수준에서는 기존 파일 시스템에 대한 참조 때와 같은 방법으로 MFS에 접근할 수 있다.

MFS에서는 파일 시스템이 제공하고 있는 기능을 이용하고 제어하기 위하여 기존의 POSIX 스타일의 API (open(), read(), write(), lseek(), close() 등) 및 파일 시스템 관리를 위한 표준적인 API(mount(), unmount(), fsck(), mkfs() 등)와 함께 아래와 같은 새로운 API들을 추가하여 응용 프로그램들이 이용할 수 있도록 하였다. 나열된 API는 실제 추가된 API들 가운데 MFS의 주요 특징과 관련된 일부이다.

▪ recycle(dev_t hdd, time_t time) 함수는 앞으로 저장될 데이터를 위한 빈 공간 확보를 위하여 인자로 주어진 시간 이전에 저장된 데이터를 모두 지운다. 즉, 이 함수는 주어진 시간에 해당하는 인덱스를 참조하여 그림 2의 메타 데이터에서, 인덱스 채널을 포함한 각 채널의 Head 와 Hoffset 값을 해당 시간의 위치로 바꾸고, 그 시간 이전까지 저장된 데이터가 차지하고 있던 블록들을 파일 시스템에 반환한다. 따라서 이 함수에서 리턴한 후, 주어진 시간 이전의 데이터는 모두 유효하지 않게 되며, 새로운 데이터를 저장하기 위한 빈 공간이 된다. 결국 이 함수는 디스크를 여러 개의 논리적인 원형 버퍼로 보이게 만든다.

▪ locate_index(int channel_id, time_t time) 함수는 MFS의 시간 기준 인덱싱 기법의 핵심적인 함수로서 인덱스 채널을 참조하여 주어진 시간 time에 해당하는 데이터 위치로 채널의 읽기 위치를 이동한다.

▪ iseek(int channel_id, struct StreamIndex *pos) 함수는 offset과 SEEK_SET 인자를 지정하여 lseek() 함수를 호출하는 것과 비슷하지만, iseek()는 파일에서의 상대적인 위치가 아닌, 주어진 절대적인 데이터 위치로 파일 읽기 위치를 이동한다. MFS에서 절대적 데이터 위치는 StreamIndex 구조체에 의해 지정되며, StreamIndex 구조체는 다음과 같이 데이터 블록 번호와 오프셋을 가지고 있다.

```
struct StreamIndex {
    int DataBlock;      /* 데이터 블록 번호 */
    int Offset;         /* 데이터 블록 내의 오프셋 */
}
```

iseek() 함수가 도입된 이유는 recycle() 함수가 각 채널 데이터의 앞부분에 있는 스트림 데이터를 삭제함으로써, recycle() 실행 후에는 채널의 데이터 시작 지점에 대한 데이터의 상대적인 위치가 변경될 수 있기 때문이다. 즉, lseek(…, SEEK_SET) 함수를 사용하는 경우, 응용 프로그램으로서는 예측할 수 없는 위치로 읽기 위치를 이동시키는 문제가 발행한다. 따라서 각 채널에서 절

대적인 데이터 위치로 이동하기 위하여 iseek() 함수를 추가하였다. 하지만 시작 위치가 아닌, 현재 읽기 위치를 기준으로 상대적인 파일 포인터 위치를 옮기는 lseek()의 경우는 MFS에서도 문제없이 사용될 수 있다.

- itell(int channel_id, struct StreamIndex *pos) 함수는 POSIX API의 ftell() 함수와 비슷하지만, 이 함수는 현재의 절대적인 파일 포인터 위치를 StreamIndex 구조체에 담아 리턴 한다.
- start_indexing(dev_t hdd) 함수는 주기적으로 인덱스 채널에 인덱스, 즉 인덱스 시점에 해당하는 각 채널의 데이터 위치, 기록을 시작한다. 만일 이 함수가 호출된 시간과 이전의 LastIndexTime (즉, 이전에 인덱싱을 정지한 시간) 사이에 시간 간격이 존재한다면, 모든 채널의 마지막 위치로 인덱스 채널의 빈 공간을 채운다.
- stop_indexing(dev_t hdd) 함수는 인덱스 정보를 기록하는 것을 정지한다.

4. 운영체제에서의 파일 시스템 구조

이 논문에서 제안된 MFS 파일 시스템은 리눅스 운영체제 상에서 우선 구현, 시험되었으며, pSOS와 같은 실시간 운영체제에도 성공적으로 이식되었다. 운영체제에서 파일 시스템의 전체 구조는 그림 4와 같다.

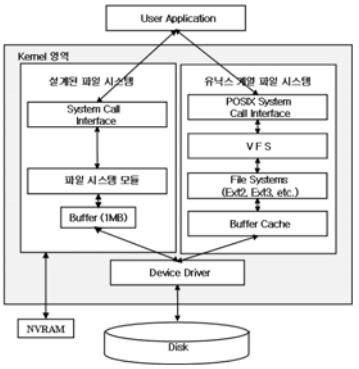

그림 4. 파일 시스템의 운영체제 상의 위치
Fig 4. MFS in Operating System

설계된 파일 시스템은 유닉스 계열의 파일 시스템들과는 달리 VFS(Virtual File System)를 배제하고 설계되어, 자체 시스템 콜 인터페이스를 제공한다. 이는 유닉스 계열 운영체제가 제공하는 VFS 환경이 가정하는 디스크 데이터 배치 구조와 설계된 파일 시스템의 데이터 배치 구조가 매우 다르기도 하며, VFS가 관리하는 버퍼 캐쉬를 사용하지 않기 때문이다. 하지만 앞서 3절에서도 언급한 바와 같이, 응용 프로그램을 위하여 VFS 환경에서 제공되는 모든 POSIX 인터페이스를 응용 프로그램에 제공함으로써 소스 프로그램 상의 호환성은 유지된다. 또한 VFS를 배제한 파일 시스템은 유닉스 계열이 아닌 많은 운영체제에도 쉽게 이식될 수 있는 장점이 있어, 실시간 운영체제 또는 운영체제 없이 운영되는 많은 임베디드 시스템에 개발된 파일 시스템을 적용할 수 있다. MFS 파일 시스템은 여러 응용 프로그램이 동시에 접근 가능하도록 만들기 위하여 상호 배제, 세마포 등 기본적인 동기화 기법들을 사용하고 있으며, 리눅스의 사용자 공간에 있는 API 라이브러리도 다중 쓰레드 프로그램에 안전하도록 구성되었다. 따라서 다중 쓰레드 환경을 지원하고, 다양한 동기화 기법들이 제공되는 대부분의 실시간 운영체제에서는 리눅스에서의 구현과 같이 여러 응용 쓰레드들이 동시에 파일 시스템에 접근하는 것이 가능하게 이식될 수 있다.

일반 유닉스 계열 파일 시스템은 디스크 블록에 대한 재사용을 고려한 성능 향상을 위해 버퍼 캐쉬를 파일 시스템과 디스크 사이에 두어 디스크를 직접 참조하지 않고 버퍼 캐쉬를 통하여 참조한다. 또 많은 운영체제에서는 시스템 내의 비할당 메모리를 버퍼 캐쉬로 적극적으로 활용하여 디스크 블록의 재사용, 쓰기의 지연 효과 등을 추구함으로써 성능 향상을 꾀하고 있지만, 순차 참조 특성으로 반복 참조가 거의 없는 멀티미디어 데이터 환경에서는 버퍼 캐쉬에 의한 성능 개선 효과가 별로 없으며, 과도한 쓰기 버퍼의 사용은 오히려 불시의 전원 차단 때 디스크 데이터의 안정성을 보장하지 못하는 중요한 원인이 된다. 따라서 구현된 MFS에서는 기존 운영체제와는 달리 사용자 공간에 있는 응용 프로그램과의 인터페이스를 위한 최소한의 쓰기 버퍼만을 유지한다. 실제 성능 평가에 사용된 DVR 시스템에서는 MFS를 구현하는데 1MB의 버퍼만을 사용하였으며, 이러한 적은 버퍼 요구는, 임베디드 시스템 환경에 개발된 파일 시스템을 쉽게 적용할 수 있도록 만든다. 또 응용 프로그램의 읽기 패턴을 분석하여 적극적인 미리 읽기를 수행하며, 데이터를 하드 디스크에 기록할 때, 수 십 msec라는 최소한의 지연 버퍼링을 한다. 특히 데이터 안정성의 확보를 위하여 인덱스 채널의 데이터와 메타 데이터는 언제나 지연 없이 디스크

에 바로 기록한다.

실제 파일 시스템 구현을 위해서는 VFS가 유지하는 여러 가지 자료 구조에 해당하는 자료 구조를 거의 그대로 유지할 필요가 있다. MFS에서도 VFS의 사용자 파일 테이블처럼, 응용 프로그램의 모든 채널 참조에 대하여 사용자 테이블 식별자를 할당하며 각각 현재 참조 위치, 사용 모드 등 필요한 정보를 별도로 유지한다. 각 사용자 테이블은, VFS의 inode 구조에 해당하는 채널 정보를 가리키고 있다. 채널 정보는 현재 각 채널의 모든 상태와 데이터 블록 매핑 정보를 가지고 있다. 버퍼 캐쉬를 기반으로 하는 VFS와는 달리 MFS의 채널 정보에는 이 채널에 대하여 현재 응용 프로그램들이 참조하거나, 쓰기가 진행 중인 모든 영역에 대한 버퍼가 링크로 연결되어 같은 채널을 참조하는 응용 프로그램들이 버퍼를 공유할 수 있도록 하고 있다. 이 버퍼는 프로그램 기동 시 할당된 버퍼 풀에서 필요할 때마다 동적으로 할당된다. 채널 정보 등의 자료 구조는 여러 개의 디스크에 있는 복수개의 MFS를 동시에 마운트하여 운용할 수 있도록 각 파일 시스템 객체마다 따로 할당된다.

여러 응용 프로그램에서의 동시 접근을 허용하는 MFS에서는, 응용 프로그램의 모든 파일 시스템 접근에 수반되는 공유 자원 접근 및 자원 확보를 위한 대기에 따른 동기화 문제가 발생한다. 따라서 사용자 테이블의 할당, 채널 정보 참조 등 주요 공유 자료 구조에 대한 접근에는 리눅스의 경우 *spin_lock()*, pSOS의 경우 *mu_lock()*, 등 상호 배제 기법을 사용하였으며, 경쟁 공유 자원인 버퍼에 대하여는 대기 큐를 이용한 sleep-wakeup 메커니즘을 사용하였다. 또 구현 환경에서 하드 디스크에 대한 참조가 동기적인 방법으로만 지원되었던 pSOS에서는, 디스크에 대한 여러 쓰레드의 동시 참조를 *sm_p()*, *sm_v()*와 같은 세마포 API를 통하여 제어하였다.

5. 파일 시스템의 안정성과 파일 시스템 복구

MFS의 중요한 설계 목표 가운데 하나는 불시의 전원 차단과 같은 비정상적 종료 후, 파일 시스템을 복구하여 다시 사용이 가능한 상태로 만들고, 복구 과정에서 데이터 손실을 최소화하는 것이다. 이 목표는 무정지 전원 장치가 사용되지 않는 가전형 멀티미디어 기기 환경에서 매우 중요하다.

MFS에서 파일 시스템 안정성의 근원은 간단한 메타 데이터 구조, 모든 메타 데이터에 대한 지연 없는 저장 그리고 배터리로 백업되는 NVRAM (Non-Volatile RAM)의 사용이다. 간단한 메타 데이터 구조는 기존 파일 시스템과 같은 복잡한 다단계 링크 구조를 배제함으로써 복구 가능성을 높이고 복구 시간을 줄이는데 도움이 되며, MFS를 구현함에 있어 모든 메타 데이터는 수정된 즉시, 디스크에 기록되어 하드 디스크가 가능한 최신 정보를 유지하도록 하고 있다. 또 그림 4에서 볼 수 있는 NVRAM은 데이터 안정성 증대를 위하여 선택적으로 사용 가능한 것으로 불시에 전원이 차단된 후나 그 밖의 다른 이유로 파일 시스템의 복구가 필요할 때를 대비하여, 주요한 메타 데이터를, 수정될 때마다 지속적으로 저장하기 위한 공간이다.

많은 경우, 파일 시스템 복구는 시스템의 메인 메모리 또는 하드 디스크 내부의 버퍼에 있던 정보가 디스크 표면에 기록되기 전에 전원이 차단된 후, 다시 부팅할 때 실시하게 된다. 실제로 디스크 표면에 기록되지 않은 정보가 주요 메타 데이터인 경우, 시스템은 상당히 많은 데이터를 참조할 수 없게 되어 해당 부분을 포기하거나, 심하면 파일 시스템 전체를 복구할 수 없는 상태에 이른다.

MFS의 경우에는 파일 시스템의 메타 데이터 구조가 매우 간단하기 때문에 NVRAM 없이도 파일 시스템 복구 가능성은 매우 높지만, NVRAM을 사용하는 경우, NVRAM에 저장된 파일 시스템 전체와 각 채널에 대한 메타데이터, 최근 인덱스 정보를 이용하여 불시의 전원 차단 이전에 기록된 데이터의 손실을 극소화할 수 있다. NVRAM에는 각 채널 정보 내의 데이터 블록 할당 맵의 처음과 마지막에 있는 몇 개의 블록 번호, 할당된 데이터 블록의 개수, 전체 데이터 크기, 그리고 각 채널의 가장 최근 인덱스들을 포함하고 있다. 또한 NVRAM과 모든 디스크 내의 메타 데이터 블록에는 체크섬을 두어 최소한의 단위 데이터 무결성을 확보하고 있다.

실제 파일 시스템 복구 프로그램은 파일 시스템 수퍼 블록 내의 주요 메타 데이터, 백업 정보, NVRAM의 정보 등 체크섬 검사에 통과한 정보들을 조합하여 디스크 전체 및 각 채널, 인덱스 데이터를 재구성하여 각 채널에 저장된 데이터 크기와 데이터 블록의 위치에 관한 가장 최신의 정보를 확보하고, 인덱스 채널의 시작과 끝 부분을 스캔하여 해당 인덱스가 각 채널의 유효한 데이터 영역을 가리키고 있는지 검사하는 과정을 거친다. 이 결과 안전하다고 확인된 부분의 데이터, 인덱스 영

역을 복구의 결과로 삼게 된다.

파일 시스템의 안정성과 복구 가능성에 관한 객관적인 실험 절차를 확보하기가 힘들어 공정한 비교 평가를 할 수는 없었지만, 여러 차례의 전원 차단 실험 결과, EXT2 파일 시스템과는 달리 MFS는 파일 시스템 복구가 100% 가능하였으며, 복구 시간도 10초 이내로 우수하였다.

6. 제안된 파일 시스템의 문제점과 극복 방안

본 논문에서 제안한 파일 시스템은 멀티미디어 스트림의 저장과 효율적인 재생을 위해 설계된 것으로 기존 파일 시스템과는 다른 특성을 가지며, 다음과 같은 문제점과 제약 사항들도 가지고 있다.

- 데이터 블록의 크기에 따른 단편화 문제 : MFS는 MB 단위의 큰 데이터 블록을 사용한다. 따라서 각 채널 데이터의 시작과 끝 데이터 블록에 각각 최대로는 데이터 블록 크기-1, 평균적으로는 데이터 블록 크기의 절반에 해당하는 저장 공간이 비어 있게 되는 단편화 문제를 가진다. 하지만 멀티미디어 데이터 스트림을 저장하는 디스크의 통상적인 크기인 수백 GB에 비하면 이 단편화에 따른 오버헤드는 상대적으로 작다고 볼 수 있다.

- 전체 저장 채널 (파일) 수의 제약 : MFS는 크기가 매우 큰, 적은 수의 연속적인 데이터 스트림을 저장할 목적으로 설계되었으며, 그들에 대한 쓰기 동작을 모니터 하고 인덱스를 유지하는 시간 기준 인덱싱 방법을 사용하고 있다. 이론 상 채널 수에 대한 제한은 없지만 인덱스 데이터의 저장, 관리 작업을 고려할 때, 전체 채널수에 대한 실용적인 제한이 생긴다. 또 인덱스 관리를 위하여, 응용 프로그램의 요구에 따라 채널수를 초기화 때 고정하도록 되어 있다. 그리고 단순한 메타 데이터 구조에 따라 디렉터리 구조에 대한 지원이 없다. 따라서 많은 작은 파일들을 가정하는 일반적인 파일 시스템 목적으로는 사용이 어렵다. 이 제약은 하드 디스크를 분할하여, 같은 물리적인 디스크 상에 MFS와 다른 일반적인 파일 시스템과 병행 운영함으로써 극복이 가능하다.

- VFS 지원 부재에 따른 제약 : 설계 목표에 따라 MFS는 VFS를 배제하고 구현되었다. 따라서 여러 장점과 함께, 바이너리 호환성 부족이라는 제약을 가지게 된다. 본 연구에서는 이 단점을 보완하기 위하여, 응용 프로그램을 재 컴파일 해야 한다는 문제가 남아있기는 하지만, MFS에 저장된 채널 데이터에의 접근이 POSIX 스타일의 AP를 통해 이루어질 수 있도록 만들어 주는 라이브러리를 제공하고 있으며 이를 통하여 소스 수준에서 기존 파일 시스템과의 호환성을 유지하고 있다.

Ⅳ. 멀티미디어 스트림 저장을 위한 파일 시스템 성능 평가

본 연구에서는 구현된 멀티미디어 스트림 저장을 위한 파일 시스템의 성능을 검증하기 위하여 MFS와 전통적인 리눅스 파일 시스템인 EXT2에 대하여, 초기화 시간과 파일 시스템 내의 여러 파일에 대한 순차 또는 임의 접근에 대한 읽기, 쓰기 및 혼합 동작에 대한 실행 시간을 측정하는 방법으로 성능 평가를 실시하였다. EXT2 파일 시스템은 비교적 단순한 리눅스의 기본 파일 시스템으로 권우일 등이 행한 연구[20] 등에서 DVR과 같은 임베디드 멀티미디어 환경에서 성능 실험을 수행한 결과, 리눅스에 적용 가능한 여러 저널링 파일 시스템들보다 성능이 전반적으로 좋은 것으로 나타났기 때문에 본 연구의 실험 대상으로 선택되었다.

1. 성능 평가 실험 환경 및 실험 방법

성능 평가를 실시한 시스템의 환경은 표 2와 같다.

표 2. 성능 평가 실험 환경
Table 2. Environment for Experiments

시스템	CPU	PowerPC (166Mhz)
	메인 메모리	32 MB (27 MB)
	운영체제 커널	Linux 2.4.2
HDD (IBM)	인터페이스	ATA/IDE
	디스크 회전 수	7200 RPM
	평균 탐색 시간	8.5 ms
	용량	40 GB

성능 평가 실험에 사용된 시스템은 실제 DVR 기기로서 5 채널의 화상, 2 채널의 오디오, 1 채널의 정지 화상 그리고 1 채널의 시스템 로그를 저장하는 데이터 채널들과, 그림 3에 기술된 바와 같이, 매 15초마다 각 채널의 데이터 위치와 해당

15초 동안 각 데이터 채널에 데이터가 저장되어있는지의 유무, 매 시간의 데이터 유무 및 인덱스 채널의 데이터 위치를 기록하는 3개의 인덱스 정보를 저장하도록 응용 프로그램이 설계된 임베디드 멀티미디어 기기이다. 실험 과정에서는 파일 시스템의 성능만을 관찰하기 위하여 화상, 오디오 등 멀티미디어 데이터 압축, 해제와 같은 기능은 정지시키고 사용자 메모리 영역의 데이터 버퍼에 저장된 더미 데이터를 기록하고, 디스크의 데이터를 사용자 데이터 버퍼에 읽는 간단한 프로그램을 작성하였다.

표 2에서 볼 수 있듯이 성능 평가에 사용된 시스템은 전형적인 소형 임베디드 멀티미디어 기기로서 메인 메모리가 32MB이고, 그 가운데 5MB를 램디스크로 사용하고 있어 실질적으로는 파일 시스템을 포함한 리눅스 커널, 공유 라이브러리, 응용 프로그램들이 사용할 수 있는 공간은 27MB에 불과하다. 따라서 위 실험 환경은 많은 메인 메모리를 가지고 있는 서버에서 사용될 것을 지향하고 있는 기존의 많은 멀티미디어 파일 시스템들이 원활하게 실행될 수 있는 환경이 아니다.

성능 평가에서는 하나의 쓰레드가 기록하는 데이터의 양을 표 3과 같이 약 1.8GB의 대용량으로 정함으로써 멀티미디어 파일 시스템의 목적에 적합하도록 하였다. 실제 DVR은 지속적으로 영상을 저장하고, 저장된 영상을 재생하는 장치로서 실제 동작 환경과 실험 환경은 차이가 있을 수 있으나, 실험에서는 1.8GB의 충분히 큰 용량의 데이터를 저장하거나 기록하고, 실제 DVR 응용에서 적용되었던 응용 프로그램 버퍼 크기 등을 이용한 실험을 진행함으로써, 실제 기기의 지속적인 참조 성능과 유사한 결과를 얻을 수 있을 것이라고 판단되었다. 실험에서 응용 프로그램이 사용하는 버퍼의 크기는 표 2와 같은 시스템 환경에서 실제 DVR 응용 프로그램이 4 채널 쓰기와 1채널 읽기를 동시에 할 때, 운영체제의 실시간 스케줄링 지원이 없는 상태에서 영상의 저장과 재생이 문제없이 이루어지는 값을 선택한 결과이다.

표 3. 성능 실험 용 부하
Table 3. Load for Experiments

응용 프로그램 버퍼 크기	128 KB
반복 횟수	15,360
단일 쓰레드의 읽기/쓰기 (버퍼크기) x (반복 횟수)	1,875 MB

또 EXT2 파일 시스템에서 성능 평가를 실시할 때에는 EXT2 자체에 시간 기준 인덱싱에 의한 접근 기능이 없으므로 EXT2 파일 시스템에 각 시간에 따른 데이터 파일에서의 위치를 기록한 별도의 인덱스 파일을 생성하고 참조하도록 하여 공정한 성능 비교 평가 실험 환경이 되도록 하였다. EXT2 파일 시스템에 사용한 인덱스 구조는 그림 5와 같다.

구현된 파일 시스템과 EXT2 모두에서 쓰기 실험이 진행되는 동안 각 단위 데이터 블록은 멀티 쓰레드 환경에서 각기 다른 시점에 각 파일 또는 채널에 할당된다. 또 그 할당이 현재 사용되지 않는 데이터 블록 맵을 참조하여 동적으로 이루어지기 때문에, 각 데이터 블록 내부의 데이터는 항상 순차적이지만, 하나의 파일 또는 채널을 구성하는 데이터 블록들의 배치는 두 파일 시스템 모두, 디스크 상에서 순차적이라고 볼 수는 없다.

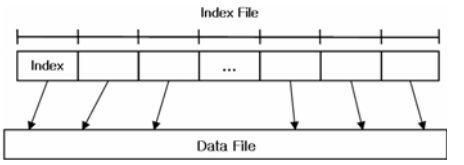

그림 5. EXT2에 사용한 시간 기준 인덱스 구조
Fig 5. Time-Based Index for EXT2

초기화 실험을 제외한 모든 성능 실험은 읽기, 쓰기 쓰레드가 동시에 실행되는 환경에서 진행되었으며, 두 파일 시스템 모두 쓰레드 별로 독립적인 파일 또는 채널 데이터에 대하여 입출력을 하도록 하였다. 또 임의 읽기에서 읽기 쓰레드는 성능 평가에 앞서서 미리 생성된 인덱스 정보를 이용하며, 쓰기 쓰레드는 인덱스 정보를 데이터 기록과 동시에 생성하여 인덱싱을 위한 부하가 성능 평가에 영향을 주도록 하였다. 성능 측정은 총 쓰레드 개수를 4개 또는 8개로 한정하고 읽기/쓰기 쓰레드의 비율을 변경하면서 실시하였다.

2. 성능 평가 결과

본 절에서는 성능 평가 결과를 보이고, 그 결과를 분석하고자 한다. 초기화 시간을 제외한 모든 성능 평가 결과는 막대그래프를 사용하여 표현하였으며, 모든 경우 같은 실험을 10회 실시한 후, 소요 시간에 대한 평균값을 결과 그래프에 사용하였다. 그래프의 Y 축은 모든 쓰레드가 실행을 완

료한 시점에 측정한 초 단위의 실행 시간을 의미한다.

2.1 초기화 시간 비교

MFS는 매우 간단한 데이터 레이아웃으로 구성되기 때문에 기존의 파일 시스템들에 비하여 초기화, 즉, 디스크 상에 초기 파일 시스템을 생성하는 작업에 드는 시간이 적다. 이 시간은 가전형 임베디드 시스템에서 새로운 하드 디스크를 설치한 뒤의 대기 시간을 의미하기 때문에, 사용자가 느끼는 시스템의 서비스 품질에 큰 영향을 끼친다.

이 초기화 실험은 표 2의 환경에서 40GB의 하드 디스크를 초기화하는데 소요되는 시간을 측정한 것이다. EXT2 파일 시스템의 경우, 의미 있는 다양한 inode 숫자에 대하여 실험을 하였다. 실제 DVR 구현에 필요한 inode의 수를 계산해보면 한 시간 단위로 디렉터리를 만들고, 각 채널에 대하여 데이터 파일과 인덱스 파일 하나씩 그리고 한 시간에 분량마다 전체 데이터 요약 정보를 생성한다고 가정할 때, 시간 당 20개의 inode가 소요되고, 한 디스크에 3개월 분량의 데이터를 저장하면 43,200개의 inode가 필요하다. 또 하드 디스크의 용량이 커져 더 오랫동안 데이터를 저장한다면 당연히 더 많은 inode가 필요하다. 실험은 위 계산에 의거하여 약 3개월 분량에 해당하는 50,000 개의 inode 수와 그 2 배, 4 배에 해당하는 inode 수에 대한 초기화 시간을 측정하였다. MFS의 경우, 디스크의 크기에 따른 초기화 시간의 차이가 없으며, 실험에서는 언제나 0.1초 이내의 시간이 소요되었다. 표 4에 있는 실험 결과에서도 MFS는 EXT2에 비하여 월등히 빠른 초기화 시간을 가진다는 것을 확인할 수 있다.

표 4. 초기화 성능 비교
Table 4. Initialization Time

MFS		0.1초 이내
EXT2 (inode 개수)	50,000개	7.4초
	100,000개	9.4초
	200,000개	16.0초

2.2 순차 참조 성능

그림 6은 두 파일 시스템의 순차 읽기와 순차 쓰기의 성능을 비교한 결과이다. 그림의 X 축에는 데이터 참조 방법과, 동시에 실행된 쓰레드의 수가 표시되어 있다. 그림 6의 결과에서 볼 수 있는 바와 같이, 순차 읽기는 쓰레드가 1개, 4개일 때 각각 22.4%, 24.9%, 순차 쓰기의 경우에는 각각 12.4%, 14.5% 차이로 MFS의 성능이 좋은 것으로 나타났다. MFS의 성능이 EXT2보다 뛰어난 중요한 이유는 디스크에 저장되는 데이터의 순차성이 훨씬 크기 때문이며, 순차 읽기의 경우, 더욱 성능이 차이가 나는 이유는 MFS의 경우, 응용 프로그램이 순차 참조를 주로 한다는 것을 고려하여 버퍼가 허용하는 한 적극적인 미리 읽기를 수행하기 때문이다. 또 쓰레드 수가 많아지면, EXT2의 경우 버퍼 캐쉬 경쟁 때문에 상대적으로 성능이 떨어지게 된다.

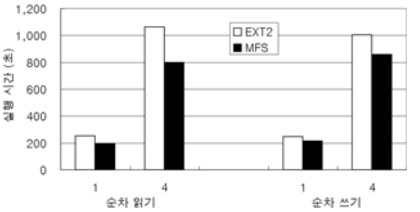

그림 6. 쓰레드 수에 따른 순차 참조 성능
Fig 6. Sequential Access Performance

2.3 인덱스 참조에 의한 임의 읽기 성능

그림 7은 두 파일 시스템에서 시간을 기준으로 구성된 인덱스를 참조하는 임의 읽기 성능을 비교한 결과이다. 임의 읽기를 구현함에 있어, MFS에서는 파일 시스템 내의 인덱스 시작 시간과 끝 시간 사이의 시간을 임의로 정해 locate_index() 함수를 통해 인덱스 채널로부터 데이터를 위치를 얻어낸 뒤 해당 위치의 데이터를 읽었고, EXT2 파일 시스템에는 MFS와 같은 시간을 기준으로 하는 인덱스가 존재하지 않으므로, lseek() 함수를 이용하여 그림 5와 같은 인덱스 파일을 임의 참조하여 데이터 위치를 얻은 뒤에, 다시 해당 위치의 데이터 파일을 참조하는 방법을 사용하였다. 그림 7에서는 MFS가 EXT2에 비하여 쓰레드가 1개일 때 35.3%, 4개일 때 46.3% 성능이 좋은 것을 보여주고 있다. 이러한 성능 개선치는 순차 읽기의 성능 차이보다 더 큰 것으로 이는 MFS가 유지하는 시간 기준 인덱싱 방법에 의한 추가적인 이득으로 해석될 수 있다. 특히 쓰레드가 4개 일 때 더 성능이 좋은 이유는 MFS는 하나의 인덱스 채널의 정보를 이용하여 모든 파일을 인덱싱 한 반면에 EXT2는 버퍼 캐쉬에 대한 경쟁과 더불어, 각 파일에 대한 별도의 인덱스를 유지함으로써 실행 시

간 동안 하드 디스크의 헤드가 더 많은 파일 사이를 움직인 결과로 해석될 수 있다.

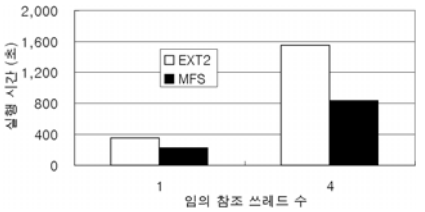

그림 7. 인덱스를 이용한 임의 참조 성능
Fig 7. Random Access Performance

2.4 순차 읽기 및 쓰기 혼합 성능

그림 8에는 순차 읽기:쓰기 쓰레드 비율에 따른 성능을 그래프로 표현하였다. 그래프의 X 축은 읽기, 쓰기 쓰레드의 비율로서, 여러 비율의 쓰레드 8개 또는 4개가 실행되는 환경에서 실험을 진행하였다. 순차 읽기, 쓰기의 경우 읽기, 쓰기 모두 순차적인 데이터 참조를 하며, 쓰기의 경우 두 파일 시스템 모두 데이터와 함께 시간 기준의 인덱스를 생성하여 기록하도록 하였다. 그림 8에서 보면 두 파일 시스템 모두 쓰레드 비율에 대한 실행 시간 편차가 그리 크지 않다. 이는 모두 순차적인 디스크 참조를 하기 때문에 하드 디스크 내부의 데이터 버퍼링, 읽기에서의 미리 읽기 기능이 잘 동작했기 때문인 것으로 판단된다.

순차 참조에 대하여 두 파일 시스템의 성능을 비교하면, MFS는 EXT2 파일 시스템에 비하여 최대 38.7%(읽기:쓰기 쓰레드 비율이 7:1인 경우)에서 최소 23.3%(읽기:쓰기 쓰레드 비율이 3:1인 경우), 평균적으로는 29.3%의 성능 향상을 보이고 있다. 이는 MFS에서는 파일 시스템의 데이터 저장 구조 상 각 단위 데이터 블록의 크기가 커서

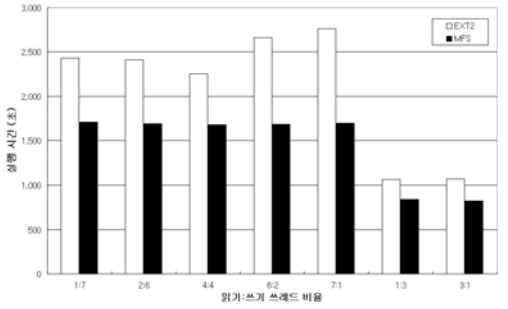

그림 8. 순차 읽기 및 쓰기 혼합 성능
Fig 8. Sequential Read and Write Performance

디스크 상의 데이터 순차성이 큼으로써, 데이터를 기록하고, 읽을 때, 디스크의 헤드 움직임이 크게 줄어든 결과이다. 또 MFS에서는 높은 순차성과 단순한 버퍼링 덕분에 읽기:쓰기 쓰레드 비율에 관계없이 쓰레드 총 개수에 따라 거의 일정한 실행 시간이 유지되는 것을 볼 수 있다.

2.5 임의 읽기 및 순차 쓰기 혼합 성능

그림 9에서는 임의 읽기:쓰기 쓰레드 비율에 따른 성능을 그래프로 표현하였다. 이 실험에서도 임의 읽기에는 2.3절의 인덱스 참조 성능 실험과 마찬가지의 방법이 사용되었다. 또 쓰기는 당연히 순차적으로 이루어졌으며, 이전 실험처럼 두 파일 시스템 모두 데이터와 함께 시간 기준의 인덱스를 생성하여 기록하도록 하였다. 임의 읽기, 쓰기에서는 인덱스 정보와 임의의 데이터 위치 참조를 위한 디스크 이동 때문에 읽기:쓰기 쓰레드 비율에 따른 실행 시간 편차가 순차 실험에서보다 매우 큰 것을 관찰할 수 있다.

이 실험에서는 MFS가 EXT2 파일 시스템에 비해서 최대 36.6%(읽기:쓰기 쓰레드 비율 이 3:1인 경우), 평균적으로는 22.3%의 성능 향상을 보이고 있다. 예외적으로, 읽기:쓰기 쓰레드 비율이 4:4인 경우에는 상황이 역전되어 EXT2 파일 시스템이 MFS보다 실행 시간이 짧은 결과를 보여주고 있는데, 이는 EXT2 파일 시스템 실험에서 이용한 시간 기준 인덱스 파일이 매우 작아, 인덱스 파일 전체가 메인 메모리의 버퍼 캐쉬에 로드되어 인덱스 기록 때 거의 실제 하드 디스크 참조를 하기 않았기 때문인 것으로 추정된다. 반면에 MFS에서는 불시의 정전 때, 데이터 복구 가능성을 높이기 위하여 모든 인덱스 정보를 매번 지연 없이 바로 디스크에 기록한다. 이런 EXT2의 버퍼 캐쉬를 이용한 이득이 실제 MFS가 가지는 데이터 순차성에 의한 이득을 능가하여 이 같은 성능 역전 현상을 만들었다. 인덱스 주기가 짧아지거나, 저장 시간이 길어 인덱스 정보가 매우 큰 경우에는, 더 이상 EXT2가 버퍼 캐쉬에 의한 이득을 누리지 못할 것으로 예상된다. 하지만 다양한 읽기, 쓰기 조합에 대한 임의 접근 실험에서도 MFS는 EXT2 파일 시스템보다 전반적으로 월등한 성능을 보여주고 있다.

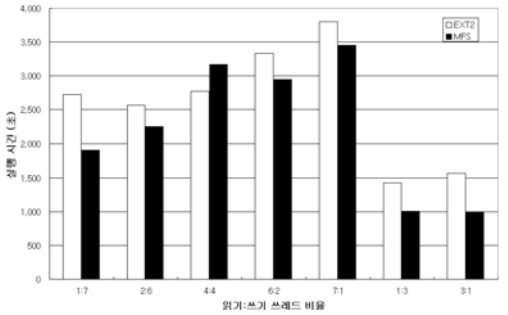

그림 9. 임의 읽기 및 순차 쓰기 혼합 성능
Fig 9. Random Read and Sequential Write Performance

2.6 성능 평가 결과 요약

본 연구에서 설계된 MFS 파일 시스템과 EXT2의 성능 비교 평가 결과, 초기화 시간, 인덱스 참조, 순차 읽기와 쓰기의 조합 실험과, 임의 읽기와 쓰기의 조합 모두에서 상당한 차이로 MFS의 성능이 좋은 것으로 확인되었다. 이는 단위 데이터 블록 크기의 증가로 인한 순차성의 증가, 응용 프로그램의 높은 순차 참조 성향을 고려한 적극적인 미리 읽기, 시간을 기준으로 하는 효과적인 인덱싱 기법에 의한 성능 이득과 , 그리고 VFS를 설계에서 배제하여 응요 프로그램에서 파일 시스템까지의 소프트웨어 계층이 엷어진 효과에 따른 추가적인 이득 때문으로 볼 수 있다.

특히 이 성능 비교 실험에서 MFS는 오직 1MB의 데이터 버퍼만을 가지고 운영되며, 모든 메타데이터, 시간 기준 인덱스에 대한 쓰기를 지연 없이 하드 디스크에 바로 한다는 점을 고려할 때, 논문에서 제안된 MFS가 기존 파일 시스템보다 멀티미디어 데이터 스트림을 처리하는데 있어 매우 우월한 성능을 가지고 있음을 여실히 보여주는 결과라고 할 수 있다.

Ⅴ. 결론과 향후 연구 과제

현재 많은 운영체제에서 사용되고 있는 기존 파일 시스템들은 대용량 멀티미디어 스트림을 처리하기 위한 임베디드 시스템에 적용하는데 있어 자원의 효율성, 멀티미디어 스트림에 대한 접근 방법의 취약성, 긴 초기화 및 파일 시스템 복구 시간 등과 같은 다양한 문제점을 가지고 있다.

이 논문에서는 대용량 멀티미디어 데이터 스트림을 저장하고, 재생하기 위한 새로운 파일 시스템을 설계하고 구현하였다. 제안된 파일 시스템은 매우 단순한 저장 구조를 가져 파일 시스템 초기화 시간이 빠르며, 저장 구조의 단순함과 NVRAM의 활용으로 파일 시스템의 안정성이 높다. 또 시간 기준 인덱싱 기법을 적용하여, 멀티미디어 데이터 스트림의 특성에 맞는 저장, 검색 및 참조 방법을 제공하고 있다. 또 설계된 파일 시스템은 운영체제 독립적인 구조로 리눅스, 실시간 운영체제 등 다양한 운영체제에 적용 가능하다.

또 본 연구에서는 설계, 구현된 파일 시스템의 성능을 평가하기 위하여 리눅스의 EXT2 파일 시스템과 다양한 읽기 쓰기 조합에 대한 성능 비교 실험을 실시하여, 최대 38.7% 성능이 우수함을 확인하였다.

본 연구에서 설계, 구현된 파일 시스템은 실제 멀티미디어 스트림을 저장하는 임베디드 시스템인 상용 다채널 DVR에 임베디드 리눅스 및 실시간 운영체제 상에서 적용됨으로써 그 유용성이 증명되었다. 하지만, 좀 더 유연한 구조로 다양한 응용에 적용할 수 있도록 파일 시스템을 개선하는 작업이 필요할 것으로 사료되며, 파일 시스템의 복구, 안정성에 관한 실험 기준과 방법에 관한 고찰을 거쳐 객관적으로 파일 시스템의 안정성을 검증할 수 있는 방법이 모색되어야 할 것이다.

참고문헌

[1] Helen Custer, Inside the Windows NT File System, Microsoft Press, 1994.
[2] Stan Mitchell, Inside the Windows 95 File System, O'Reilly, 1997.
[3] -, Volume and File Structure of CD-ROM for Information Interchange, ISO/IEC-9660, ISO, 1999.
[4] -, Universal Disk Format (UDF) Specification, Rev. 2.6, Optical Storage Technology Association, 2005.
[5] Card, R., Ts'o, T. and Tweedie. S., "Design and Implementation of the Second Extended Filesystem", In Proceedings of the First Dutch International Symposium on Linux, 1994.
[6] Michael K. Johnson, "White paper: Red Hat's New Journaling File System: ext3", Red Hat, http://www.redhat.com/support/wpapers/redhat/ext3/index.html#toc

[7] -, Journaled File System Technology for Linux, html document of IBM, http://oss.software.ibm.com/jfs/
[8] -, XFS html document of SGI, http://oss.sgi.com/projects/xfs/
[9] Bryant, R., Forester, Ruth and Hawkes, John, "Filesystem Performance and Scalability in Linux 2.4.17", In Proceedings of 2002 USENIX Conference, pp. 259-274, 2002.
[10] T. N. Niranjan, T. Chiueh, G.A. Scholoss, "Implementation and Evaluation of a Multimedia File System", In Proceedings of International Conference on Multimedia Computing and Systems, pp. 269-276, 1997.
[11] R.L. Haskin, F.B. Schmuck, "The Tiger Shark file system", In Proceedings of the IEEE COMPCON, pp. 12-15, 1996.
[12] C. Martin, P.S. Narayan, B. Ozden, R. Rastogi, and A. Silberschatz, "The Fellini Multimedia Storage System", Journal of Digital Libraries , 1997.
[13] M.M. Buddhikot, X.J. Chen, D. Wu, G.M. Parulkar, "Enhancements to 4.4 BSD UNIX for Efficient Networked Multimedia in Project MARS", In Proceedings of IEEE International Conference on Multimedia Computing and Systems, pp. 326-337, 1998.
[14] M.S. Chen, D.D. Kandlur and P.S. Yu, "Optimization of the Grouped Sweeping Scheduling (gss) with Heterogeneous Multimedia Streams". In Proceedings of ACM Multimedia, pp. 235-242, 1993.
[15] D.R. Kenchammana-Hosekote and J. Srivastava, "Scheduling Continuous Media on a Video-On-Demand Server", In Proceedings of International Conference on Multimedia Computing and Systems, pp. 19-28, 1994.
[16] P.J. Shenoy, P. Goyal, S. Rao, and H.M. Vin, "Design Considerations for the Symphony Integrated Multimedia File System", ACM/Springer Multimedia Systems Journal, pp. 337-352, 2003.
[17] B.-S. Ahn, S.-H. Sohn, C.-Y. Kim, G-I. Cha, Y.-C. Baek, S.-I. Jung and M.-J. Kim, "Implementation and Evaluation of EXT3NS Multimedia File System", In Proceedings of the 12th annual ACM international conference on Multimedia, pp. 588-595, 2004.
[18] Mitchell, Pennebaker, Fogg, and LeGall, MPEG Video Compression Standard, Springer, 1996.
[19] Pereira and Ebrahimi, The MPEG-4 Book, Prentice Hall PTR, 2002.
[20] 권우일, 윤미현, 이동준, 장재혁, 양승민, DVR 시스템을 위한 저널링 파일 시스템의 성능평가, 2002 추계 한국정보과학회 논문지, pp. 397-399, 2002.

저 자 소 개

이 민 석

1986, 88, 95년 서울대학교 컴퓨터공학과 학사, 석사, 박사. 1999-2002년 (주)팜팜테크 CTO, 1995년 이후 현재까지 한성대학교 컴퓨터공학과 교수 재직 중. 관심분야: 임베디드 시스템, 공개 소스, 파일 시스템, 임베디드 리눅스

Email: mslee@hansung.ac.kr

송 진 석

2005년 한성대학교 컴퓨터공학과 학사. 현재 동과 석사과정 재학 중. 관심분야: 임베디드 시스템, 파일 시스템, 스케줄링 알고리즘

Email: kopuk@hansung.ac.kr