임베디드 시스템을 위한 파일 시스템 구현

강석민,°송재영, 조정철, 권택근
충남대학교 컴퓨터공학과

Implementation of File System for Embedded System

Seok-Min Kang°, Jae-Young Song, Jung-Chul Jo, Taeck-Geun Kwon
Dept. of Computer Engineering, Chungnam National Univ.
{smkang, jysong, jcjo, tgkwon}@ce.cnu.ac.kr

요 약

컴퓨터 및 네트워크 기술의 눈부신 성장은 PDA, MP3 플레이어, 디지털 카메라와 같은 임베디드 시스템의 급성장을 가져왔다. 이러한 임베디드 시스템에는 그 시스템의 목적에 맞도록 특화된 실시간 운영체제가 탑재되게 되고, 그에 맞게 각 저장 장치들을 제어할 수 있는 파일 시스템도 필요하다.

본 논문에서는 삼성전자에서 개발한 CalmRISC16 마이크로 프로세서 코어를 사용하는 임베디드 시스템에 탑재될 실시간 운영체제를 위한 임베디드 파일 시스템을 구현하였다. 본 논문에서 구현된 임베디드 파일 시스템은 가상 파일 시스템으로 동작하며 In-memory 파일 시스템과 FAT를 사용하는 SmartMedia를 지원한다.

1. 서 론

컴퓨터 하드웨어 기술과 인터넷을 포함한 정보 통신의 발달은 사무 작업, 교육 및 휴련, 정보 검색 등의 다양한 서비스를 제공해 주고 있는 개인용 컴퓨터의 보급을 촉진시켰다. 이러한 개인용 컴퓨터의 고속 성장은 개인 휴대 단말기와 같은 임베디드 시스템에도 영향을 미쳤다. 이처럼 개인용 컴퓨터 시장의 포화 현상, 개인용 휴대 장비의 다양화와 이동형 네트워크 (Mobile Network)가 맞물려 임베디드 시스템의 중요성이 부각되고 있다.

임베디드 시스템은 그 시스템의 목적에 맞게 특화된 운영체제와 응용 프로그램을 탑재하게 된다. 임베디드 시스템용 운영체제에 포함될 파일 시스템도 이러한 목적에 부합하는 기능을 지녀야 한다. 또한 임베디드 시스템은 휴대성이라는 특성 때문에 내부 저장 공간이 부족할 수 밖에 없고 이런 단점을 극복하기 위해 다양한 외부 저장 장치를 지원해야 한다.

본 논문에서는 삼성 전자에서 개발한 CalmRISC16 마이크로 프로세서 코어를 사용하는 임베디드 시스템에 적합한 임베디드 파일 시스템 (EFS : Embedded File System)을 구현하였다. EFS는 가상 파일 시스템으로, 메모리를 저장 공간으로 사용하는 In-memory 파일 시스템과 FAT를 사용하는 SmartMedia 메모리 카드를 지원한다.

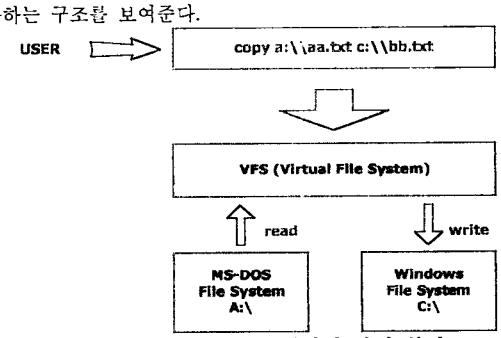
본 논문의 구성은 다음과 같다. 2장에서는 본 논문에서 구현한 임베디드 파일 시스템의 구조와 동작에 대해 소개한다. 3장에서는 임베디드 파일 시스템에서 지원하는 저장 장치인 SmartMedia에 대해서 기술한다. 4장에서는 파일 시스템과 디바이스 드라이버 등의 구현 내용을 기술하고 결론 및 향후 연구 과제는 5장에서 기술한다.

2. 임베디드 파일 시스템의 구조와 동작

파일 시스템은 파일을 생성할 때 물리적 저장 공간을 할당하고, 삭제할 때 해제된 빈 공간을 관리하는 등의 작업을 처리해 준다. 본 논문에서 구현한 EFS는 이러한 파일 처리 작업을 메모리와 SmartMedia를 대상으로 할 수 있도록 되어 있다.

2.1 가상 파일 시스템

일반적으로 파일 시스템은, 여러 종류의 저장 장치들을 지원하게 된다. 이러한 여러 저장 장치에 사용자가 파일을 저장할 경우, 각 저장 장치에서 사용하는 파일 시스템에 상관없이 통일된 인터페이스를 통해 작업을 하도록 하고 실제 동작은 파일 시스템이 저장 장치에 적합한 함수를 호출하도록 해주는 것이 가상 파일 시스템이다. EFS는 이러한 가상 파일 시스템의 역할을 하게 된다. 다음의 [그림 1]은 가상 파일 시스템에서 사용자가 서로 다른 저장 장치간에 파일을 복사하는 구조를 보여준다.



[그림 1] 가상 파일 시스템에서의 파일 복사

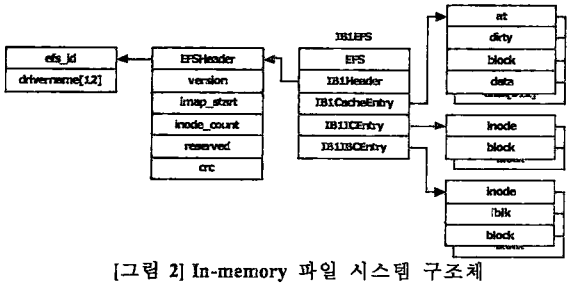
2.2 In-memory 파일 시스템

EFS는 시스템의 메모리를 저장 공간으로 사용하는 In-memory 파일 시스템을 지원한다. 앞에서 언급했듯이 임베디드 시스템은 휴대성이라는 특성으로 인하여 충분한 외부 저장 공간을 확보하기 어려운 경우가 많기 때문에, 시스템에 내장된 메모리를 활용할 수 있는 In-memory 파일 시스템을

구현하였다. 이번 장에서는 In-memory 파일 시스템을 예로 하여 EFS 파일 시스템과 파일의 구조에 대해 소개한다.

2.2.1 In-memory 파일 시스템 구조

In-memory 파일 시스템은 다른 파일 시스템과 마찬가지로 파일과 디렉토리를 다루기 위한 구조체, 최근에 접근했던 파일에 대한 정보를 저장하는 캐쉬, 각 파일에 대한 아이노드 테이블을 관리한다. [그림 2] 는 이러한 In-memory 파일 시스템에서 다루는 전체 구조체를 보여준다.



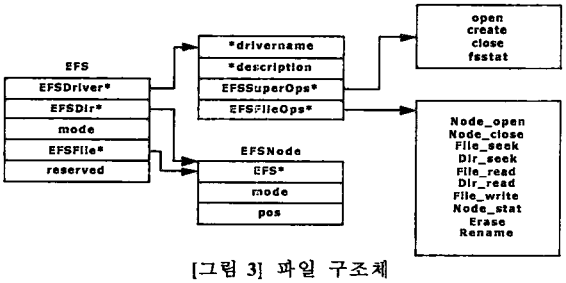
[그림 2] In-memory 파일 시스템 구조체

파일 시스템은 IB1EFS라는 파일 시스템당 한 개의 구조체를 유지한다. IB1EFS의 IB1Header를 보면 그 파일 시스템의 버전, ID, 드라이브명 등을 유지하고 있고 접근했던 파일의 정보에 대한 캐쉬를 저장하는 캐쉬 엔트리 리스트도 유지하고 있다. [그림 2]에서 보듯이 EFS는 아이노드를 사용하여 파일을 관리한다. 아이노드는 파일이 실제 저장 공간의 어느 블록에 저장되는지에 대한 정보를 유지하고 있기 때문에 파일에 대한 연산들이 아이노드를 통하여 이루어 질 수 있다.

또한, EFS에서는 세 종류의 캐쉬를 관리한다. 먼저 CacheEntry는 일반적인 캐쉬로서 파일에 접근하여 처리했던 데이터를 저장해 둠으로써 효율을 높인다. ICache는 아이노드 캐쉬로서 최근에 접근한 아이노드의 정보와 그 아이노드의 시작 블록이 몇번인지를 저장해 둔다. IBCEntry는 ICache에 저장된 아이노드가 가진 블록의 수가 캐쉬크기보다 클 때 그 나머지 블록에 대한 정보를 저장하는 인다이렉트 캐쉬이다.

2.2.2 In-memory 파일 구조

다음의 [그림 3]은 EFS에서 사용하는 파일에 대한 구조체이다.



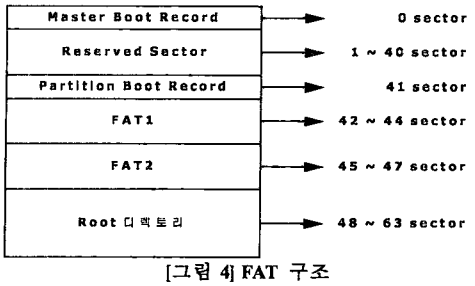
[그림 3] 파일 구조체

각 파일은 EFS - In-memory 일 경우 IB1EFS - 라 불리는 구조체를 가지게 된다. [그림 3]에서 볼 수 있듯이 EFS 구조체는 이 파일이 저장된 저장 장치가 사용하는 파일 시스템이 어느 것인지에 대한 정보를 가지는 EFSDriver, 이 파일이 일반 파일인지 디렉토리인지에 대한 정보를 가지는 EFSFile(EFSDir), 파일이 읽기 전용인지, 편집 가능인지의 정보인 mode를 가지고 있다. EFSDriver 포인터를 따라가 보면 파일이 저장된 저장 장치를 알 수 있고, 이 저장 장치에 적합한 파일 연산들을 사용할 수 있게 된다. 예를 들어, 저장 장치가 메모리라면 In-memory 파일 시스템을 사용해야 하므로 In-memory 파일 시스템에 적합한 open/close/read/write 의 파일 연산들이 수행되게 된다. [그림 3]에서 SuperOps는 파일 시스템 자체를 다루는 연산들이고 FileOps는 그 파일 시스템의 파일에 대한 연산들이 연결되어 있다.

3. SmartMedia

SmartMedia는 우표 크기의 작은 메모리 카드이다. 초기에 SSFDC(Solid State Floppy Disk Card)라 명명되었다가 1996년 10월 SSFDC 포럼에서 SmartMedia라는 명칭으로 통일하였다. SmartMedia는 카드 내에 제어칩을 탑재하고 있지 않기 때문에 2g 정도로 가볍고 45.0mm x 37.0mm x 0.67mm 의 크기로 매우 작기 때문에 이동형 기기에 적합한 저장 매체이다.

SmartMedia는 개인용 컴퓨터와의 호환을 위해 FAT(File Allocation Table) 운영체계를 사용한다. FAT는 MS-DOS나 Windows 같은 OS에서 널리 사용하는 범용적인 운영체계이다. FAT는 저장 장치내에 유지되는 파일 배치표로써, 파일들이 저장되어 있는 섹터에 대한 위치도를 제공해 주는 역할을 한다. SmartMedia는 매체의 용량에 따라 다른 종류의 FAT를 사용한다. 64MB까지의 SmartMedia는 FAT12를, 128MB 이상의 SmartMedia는 FAT16을 사용하는데 본 논문에서 구현한 EFS는 128MB SmartMedia는 고려하지 않았다. [그림 4]는 FAT의 구조을 보여준다.



[그림 4] FAT 구조

Master Boot Record는 부트 코드로써 시스템 BIOS 개념으로 사용된다. FAT에서 Master Boot Record는 절대 위치인 0 섹터에 쓰여진다. Master Boot Record에는 기본 부트 파티션에 대한 정보가 저장되어 있다. Partition Boot Record에는 기본 파라미터에 대한 정보가 저장되어 있다. BIOS 파라미터 블록, 레이블, FAT 구조등이 Partition Boot Record에 포함된다. FAT1에는 파일들의 연결 리스트에 대한 정보가 저장되고, FAT2는 FAT1이 손상되었을 때를 위한 백업 용도로 사용된다.

4. 구현 내용

본 논문에서 구현한 임베디드 파일 시스템은 Intel CPU를 사용하는 개인용 컴퓨터에서 1차적으로 테스트 되었다. 컴파일러는 Microsoft사의 Visual C++ 6.0을 사용하였다. 2차 버전은 삼성 전자에서 개발한 CalmRISC16 마이크로 프로세서 코어를 사용하는 보드에 포팅하여 테스트 하였다. 컴파일러는 CalmRISC용 통합 개발 환경인 CalmSHINE을 사용하였고 디바이스 드라이버의 하위 부분은 어셈블리, 파일 시스템의 다른 부분은 C 언어를 사용하였다.

4.1 표준 인터페이스

EFS는 가상 파일 시스템으로 동작을 하며, 사용자에게 일

정한 인터페이스를 제공해주고 내부적으로는 접근하는 파일 시스템에 적합한 함수를 호출하여 파일 I/O 연산을 수행하게 된다. 본 논문에서 구현된, EFS가 사용자에게 제공하는 표준 인터페이스, 즉 API는 다음 [표 1]과 같다.

[표 1] EFS가 제공하는 API

	EFSFile *fopen(const char *path, const char *mode)
	path의 파일을 여는 API
	int fseek(EFSFile *file, long offset, int whence)
	파일내의 포인터를 조정하는 API
	int fclose(EFSFile *file)
	지정한 file을 닫는 API
	int fputc(int c, EFSFile *file)
	c를 file에 쓰는 API
	int fgetc(EFSFile *file)
	file에서 한 바이트를 읽어오는 API
	int fstat(EFSFile *node, EFSStat *stat)
	node의 정보를 반환하는 API
	size_t fread(void *ptr, size_t size, size_t mmemb, EFSFile *file)
	file에서 size x mmemb 만큼을 읽어오는 API
	size_t fwrite(void *ptr, size_t size, size_t mmemb, EFSFile *file)
	file에 size x mmemb 만큼 쓰는 API
	int unlink(const char *pathname)
	pathname의 파일을 지우는 API

4.2 쉘

임베디드 시스템의 경우 사용자와의 상호 작용을 위해 직렬 또는 병렬 포트를 통한 콘솔, LCD 디스플레이 등을 제공한다. EFS가 테스트 된 보드에 실제 디스플레이가 있지는 않지만 추후에 제공되고 시스템 디버깅 및 테스트를 고려하여 별도의 쉘을 구현하고 파일과 관련된 몇가지 명령어들도 구현하였다. 쉘을 구동하면 프롬프트가 나타나고 몇가지 명령어들을 이용하여 파일의 복사, 삭제, 디렉토리 생성 등의 작업을 할 수 있다. [표 2]는 본 논문에서 구현한 파일 관련 명령어들이다.

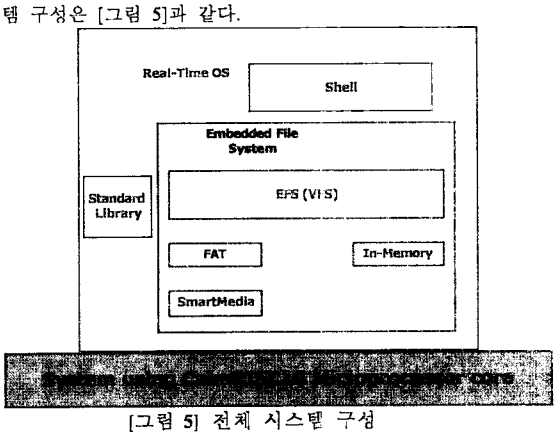
[표 2] 파일 관련 명령어

cat	파일의 내용을 표준 출력에 보여주는 명령어
cd	현재 작업 디렉토리를 변경하는 명령어
cp	파일을 복사하는 명령어
free	메모리 사용 상황에 대한 정보를 출력하는 명령어
ls	현재 경로의 파일 리스트를 출력하는 명령어
mkdir	새로운 디렉토리를 생성하는 명령어
mv	파일을 바꾸는 명령어
pwd	현재 작업 디렉토리의 경로를 출력하는 명령어
rm	파일을 삭제하는 명령어
rmdir	디렉토리를 삭제하는 명령어
stat	파일 시스템에 대한 정보를 출력하는 명령어

4.3 SmartMedia 디바이스 드라이버

In-memory 파일 시스템을 위한 파일 I/O 함수들은, 시스템의 메모리를 저장 공간으로 사용하기 때문에 새로운 파일이 생성될 때 운영체제로부터 필요한 메모리 영역을 할당받아 파일의 내용을 저장하게 된다. 이때 새로운 메모리 영역을 할당받는 방법은 운영체제가 제공하는 malloc() 호출을 사용하는 것이다. 그러나 SmartMedia의 경우 FAT를 사용하고, 메모리와는 달리 운영체제가 지원해 주는 함수가 없기 때문에 어셈블리어를 이용한 하위 레벨 I/O 모듈을 작성하고, 상위 레벨에서는 어셈블리로 쓰여진 함수를 이용하는 API들을 구현하였다. 상위 레벨의 API는 [표 1]과 인터페이스가 동일하다.

4장에서 살펴본 구현 내용을 토대로한 전체 임베디드 시스템 구성은 [그림 5]과 같다.



[그림 5] 전체 시스템 구성

5. 결론 및 향후 연구 과제

본 논문에서는 임베디드 시스템을 위한 파일 시스템을 구현하였다. 이 파일 시스템은 가상 파일 시스템으로 사용자에게 표준적인 인터페이스를 제공하고, 하위 계층에서 In-memory 지원, FAT를 통한 SmartMedia를 지원한다.

향후 연구 과제로는, 보다 다양한 디바이스를 지원할 수 있도록 디바이스 드라이버의 추가 구현이 필요하다. 또한 임베디드 시스템은 자원의 한계가 필연적이기 때문에 적은 자원을 보다 효율적으로 사용할 수 있는 성능 개선에 대한 연구도 필요하다.

참고 문헌

[1] M. Rosenblum and J. K. Ousterhout, "The Design and Implementation of a Log-Structured File System", ACM Transactions on Computer Systems, Vol.10, No.1, 1992.

[2] M. K. McKusick, M. J. Karels, and K. Bostic, "A Pageable Memory Based File System", Proc. '90 Summer USENIX, 1990.

[3] M. Wu and W Zwaenepoel, "eNVy: A Non-Volatile, Main Memory Storage System", Proc. 6th International Conference on Architectural Support for Programming Languages and Operating Systems, 1994.

[4] R. McGrath and R. Stallman, "The GNU C Library Reference Manual", 1996.

[5] WindRiver, http://www.wrs.com, 2001.

[6] microC/OS, http://www.ucos-ii.com, 2001.

[7] The KU Real-Time Linux, http://www.ittc.ku.edu/kurt/, 2001.

[8] RedHat Embedded Linux, http://www.redhat.com/embedded, 2001.

[9] SSFDC Forum, http://www.ssfdc.or.jp/english/index.htm, 2001.