리눅스 기반의 사용자 수준 플래시 파일 시스템의 구현

권 우 일[†] · 박 현 희[††] · 양 승 민[†††]

요 약

리눅스를 운영체제로 사용하는 PDA, 전자수첩 등의 소형 임베디드 시스템의 사용이 증가하고 있다. 그러나 리눅스 커널은 모노리딕(monolithic)하다는 특성 때문에 다양한 형태의 임베디드 시스템에 필요한 요구사항을 만족하지 못하고 있다. 본 논문에서는 모노리딕 커널의 단점을 보완하기 위해 리눅스에서 널리 사용되는 JFFS 파일 시스템을 커널에서 분리하여 사용자 수준에서 응용 프로그램 프로세스로 실행되는 uJFFS를 구현한다. uJFFS는 파일 시스템과 플래시 디바이스 드라이버로 구성되며 커널과 분리되어 실행되므로 커널을 소형화할 수 있다. uJFFS는 파일 시스템의 자료구조가 사용자 주소 공간에 존재하며, 파일 시스템을 담당하는 ujffs_fs와 플래시 디바이스를 제어하는 ujffs_drv 드라이버로 구성된다. 또, uJFFS는 기존의 리눅스에서 제공하는 것과 동일한 인터페이스를 지원한다. 물리장치에 접근하기 위한 디바이스 드라이버 역시 사용자 영역에서 구현되어 장치나 파일 시스템의 오류가 발생하더라도 커널에 미치는 영향을 최소화하여 시스템의 안정성을 증가시킬 수 있다.

Implementation of The User-level Flash File System Based on Linux

Woo Il Kwon[†] · Hyun Hui Park[††] · Seung Min Yang[†††]

ABSTRACT

The number of applications of small embedded systems such as PDAs, electronic note books, etc. based on Linux, have increased. Due to the monolithic characteristic of Linux kernel, it is not suitable to satisfy the various kinds of embedded application requirement. To assist the shortcoming of monolithic kernel, we implement uJFFS flash file system as an application program process which runs in user space. This solution consists of a file system and a flash device driver, and makes Linux kernel smaller by separating the file system from the kernel. uJFFS consists of ujffs_fs that plays a part of file system and ujffs_drv that controls a flash device. Which provides the same user interface as Linux does. A Device driver for the physical device is implemented in user space, which prevents kernel failures from file system errors. So uJFFS can increase stability of the system.

키워드 : 운영체제(Operating System), 플래시 파일 시스템(Flash File System), 리눅스(Linux), 임베디드 시스템(Embedded System)

1. 서 론

리눅스를 셋탑박스, PDA, 스마트폰 등의 한정된 자원만을 이용할 수 있는 소형 정보 가전기기에 탑재하기 위해서는 커널의 소형화가 필수이다. 그러나 리눅스 커널은 원래 임베디드 시스템을 위한 운영체제로 설계되지 않았으며, 커널이 모노리딕(monolithic)하다는 특성 때문에 임베디드 시스템에 응용에 맞도록 항상 커널 설정을 재구성해야 한다. 따라서 다양한 형태의 임베디드 시스템에 바로 적용하기가 어렵기 때문에 제한된 자원을 이용하는 소형 장비에는 적합하지 않다. 소형 임베디드 시스템은 범용 시스템과는 달리 저장 장치를 필요로 하지 않는 시스템도 있으며, 저장 장치를 필요로 하는 소형 장치들 역시 시스템의 동작을 위해서 최소한의 저장 장치와 파일 시스템을 필요로 한다. 현재 리눅스 기반의 소형 장비들에서는 저장 장치로 플래시 메모리가 널리 사용되고 있으며, 플래시 메모리를 위한 파일 시스템으로는 JFFS, JFFS2, yaFFS등 여러 가지가 있다. 이러한 파일 시스템은 모노리딕 특성을 가진 리눅스 커널의 일부로 커널 영역에서 동작하며 응용 프로그램의 입출력 요청을 서비스하기 위해서는 반드시 최소한의 커널 구성에 포함되어야 한다.

본 논문에서는 앞에서 지적한대로 모노리딕한 리눅스 커널의 단점을 개선하여 플래시 메모리 기반의 저장장치를 필요로 하지 않는 시스템을 위해 종래에 운영체제의 일부로 존재하는 JFFS 플래시 파일 시스템과 디바이스 드라이버를 커널에서 분리하여 사용자 수준에서 응용 프로그램 프

※ 본 연구는 숭실대학교 교내 연구비 지원으로 이루어졌습니다.
† 정 회 원 : 한국전자통신연구원 임베디드S/W기술센터 연구원
†† 준 회 원 : 숭실대학교 대학원 컴퓨터학과
††† 정 회 원 : 숭실대학교 컴퓨터학부 교수
논문접수 : 2004년 1월 15일, 심사완료 : 2004년 5월 4일

로세스로 동작하는 uJFFS 파일 시스템을 구현한다. 이것은 리눅스 커널에 마이크로 커널의 개념을 도입하여 종래의 커널의 일부로 동작하는 플래시 파일 시스템과 플래시 드라이버를 사용자 영역에서 동작하는 응용 프로그램으로 구현하여 파일 시스템을 리눅스 커널에서 쉽게 분리할 수 있도록 하는데 그 목적이 있다.

uJFFS는 플래시 메모리(flash memory) 기반의 파일 시스템이다. 플래시 메모리는 회전식 자기 매체를 이용한 디스크에 비해 외부 충격에 강하고, 비휘발성이다. 따라서 불안정한 환경에서도 동작해야하는 임베디드 시스템에 적합하다. 또, 기계적으로 회전하는 디스크와 같은 매체에 비해 빠른 속도로 접근할 수 있는 메모리 장치이며, 저 전력으로 구동이 가능하고, 크기가 작아서 소형 내장 시스템의 보조 기억 장치로서 하드디스크를 대체하고 있다. 그러나 이러한 장점 외에 물리적인 특성으로 인해 몇 가지 문제점이 존재한다. 첫째는 한 번 데이터를 쓴 메모리 영역에 바로 다시 쓰기가 불가능하며, 새로운 데이터를 쓰기 위해서는 그 전에 반드시 메모리를 지워야 한다. 둘째로 메모리를 지울 수 있는 회수가 100,000번 정도로 제한되어 있어, 영구적으로 사용할 수 없기 때문에, 지우는 작업을 최대한 줄일 수 있는 정책을 사용하여 수명을 길게 해주어야 한다.

uJFFS의 구현은 다음과 같은 점에 중점을 둔다. 첫째, 기존의 LFS 기반의 저널링 기능을 지원하는 JFFS를 원형으로 하는 파일 시스템과 플래시 드라이버가 사용자 영역의 응용 프로세스로 존재하며, 다른 사용자 프로세스의 파일 서비스 요청에 대해 동작한다. 따라서 커널 영역에서 동작하는 기존 파일 시스템과는 달리 사용자 영역에서 파일 시스템 자료구조를 유지하면서 요청을 서비스한다. 둘째, 리눅스 커널의 수정을 최소화 한다. 이를 위해서 커널 내부에는 사용자 수준의 파일 시스템과 디바이스 드라이버에 커널 인터페이스를 지원하기 위한 모듈을 사용한다. 유닉스 호환의 운영체제는 VFS(Virtual File system Switch) 계층이 존재한다. VFS란 표준 유닉스 파일 시스템과 관련된 모든 시스템 콜을 처리하는 소프트웨어 계층이다[7]. 리눅스 시스템에 공존하고 있는 모든 파일 시스템은 VFS 가상 계층에 의해 추상화 되어 사용자에게는 저장된 파일이 가상 파일 시스템 계층에 의해 공통 파일 모델로 표현되어 동일한 사용자 API로 조작할 수 있다. 사용자 수준 JFFS는 VFS의 구조를 수정하지 않으며, 따라서 사용자에게는 기존 파일 시스템과 동일한 인터페이스를 제공한다.

본 논문의 구성은 다음과 같다. 2장에서는 사용자 수준에서 운영체제 서비스를 지원하기 위해 먼저 시도된 연구들을 살펴본다. 3장에서는 리눅스 커널에서 파일 시스템이 사용자 수준에서 구현되기 위해 필요한 요구사항과 메커니즘을 설명하고, 4장에서는 논문에서 제안하는 사용자 수준 파일 시스템의 자료 구조와 동작 방법, 구현에 대해서 설명한다. 5장에서는 uJFFS와 기존 JFFS 파일 시스템과의 비교를 통한 성능 평가에 관해 언급하고, 마지막으로 6장에서는 결론과 향후 연구방향을 제시한다.

2. 관련 연구

2.1 리눅스에서의 사용자 수준 운영체제 서비스

Userdev는 앞에서 말한 리눅스의 단점을 보완하여 Mach 등의 마이크로 커널과 유사하게 리눅스에서 사용자 수준 디바이스 드라이버를 구현할 수 있는 라이브러리를 제공하여 기존 디바이스 드라이버의 작성과 유사한 인터페이스를 사용자 수준에서 사용할 수 있다[1, 4, 5]. Userdev 라이브러리를 사용하여 구현된 사용자 수준 디바이스 드라이버는 개발 시 다양한 사용자 라이브러리를 이용할 수 있으며, 드라이버에 오류가 발생하더라도, 커널에는 영향을 미치지 않는다. 현재는 Sprite에서 사용된 의사-디바이스(pseudo-device)와 유사하게 Userdev를 이용하여 의사-터미널(pseudo-terminal)과 같은 의사-디바이스와 램 디스크 드라이버를 구현할 수 있으며, 원격 디바이스에 투명하게 접근할 수 있다[9].

FUSD는 사용자 공간에서 캐릭터 디바이스를 제어할 수 있는 프레임워크(framework)를 제공한다. 사용자 수준 드라이버를 위한 라이브러리와 커널 모듈로 구성되어 있으며, /dev에 다른 장치들과 마찬가지로 존재하며, 커널 콜백 함수에 의해 요청을 수행한다. 또 커널 수준의 드라이버와는 달리 FUSD를 사용해서 구현된 디바이스 드라이버는 네트워크, 시리얼 포트, 윈도우등 다른 디바이스 드라이버에도 사용될 수 있다[12]. 그러나 FUSD는 캐릭터 디바이스만을 지원하기 때문에 플래시 메모리와 같이 블록 디바이스로 간주되는 디바이스에 대해서 적용하는 것이 불가능하며, 파일 시스템과 같은 논리적인 자료구조를 구현할 수 없다.

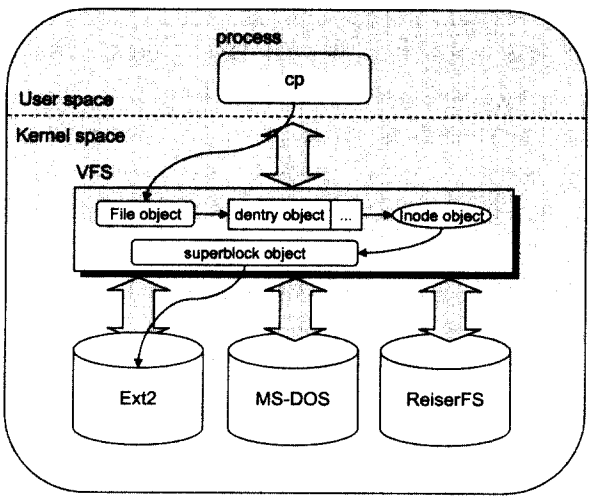
Userfs는 리눅스에서 사용자 수준에 파일 시스템을 구현하기 위한 시도로, 파일 시스템을 일반 사용자 프로세스로 구현한 것이다. 새로운 파일 시스템의 시험 버전이나, 자주 사용되지 않는 파일 시스템, 존재하는 파일 시스템에 부가적인 기능을 확장하기위한 용도, 또는 완전한 가상 파일 시스템에 적용할 수 있다[10, 11]. 그러나 새로운 파일 시스템을 개발한다는 것은 매우 복잡한 작업이며, 이미 리눅스에 존재하는 VFS와 연동하는 것이 힘들다는 단점이 있다.

사용자 영역에 동작하는 이러한 운영체제 서비스들은 다른 사용자 프로세스의 요청을 커널을 통해 전달받아서 동작하며, 프로세스간 통신은 커널의 IPC(Inter Process Communication)를 이용한다. 사용자 수준의 드라이버를 이용한 방법은 파일 시스템과 같은 운영체제 서비스가 커널의 외부에 위치하기 때문에 기존 모노리딕 커널에 비해 잦은 문맥 교환 회수와 제어의 흐름이 복잡하여 성능 저하가 발생

한다. 따라서 문맥 교환 속도를 빠르게 해야 하며, 불필요한 데이터의 흐름을 방지 할 수 있도록 설계되어야 한다[6].

2.2 가상 파일 시스템(Virtual File system Switch)

리눅스는 여러 가지 종류의 파일 시스템이 동시에 공존할 수 있다. 이는 리눅스에서 동작하는 모든 파일 시스템이 다른 유닉스 시스템과 유사하게, 사용자에게는 동일한 방법으로 접근할 수 있도록 하는 가상 파일 시스템을 지원하기 때문이다. 리눅스 가상 파일 시스템(VFS, Virtual File system Switch)은 표준 유닉스 파일 시스템과 관련된 시스템 콜을 처리하는 소프트웨어 계층이다. VFS는 서로 다른 구조를 가진 파일 시스템을 공통 파일 모델(Common File Model)이라는 구조로 정의하여 사용자에게는 모든 파일 시스템이 동일한 구조로 보이도록 한다. 파일 시스템이 마운트 된다는 것은 파일 시스템의 슈퍼블록(superblock)을 읽어서 VFS의 superblock 구조체에 내용을 기록하는 것이다. 한편 파일 시스템은 자신의 물리적인 구조를 VFS의 공통 파일 모델로 변환할 수 있어야 한다. 공통 파일 모델은 superblock, inode, file, dentry와 같은 객체로 구분되며, VFS에서는 객체를 처리하는 파일 오퍼레이션을 정의한다. 이들은 각각의 시스템 콜과 동일한 이름의 함수로 구성되며, 특정 파일 시스템과 응용 프로그램 사이에 관련된 시스템 콜을 처리한다[7]. (그림 1)은 사용자 프로세스에서 cp 명령이 수행될 때 VFS 객체를 차례로 참조하여 최종적으로 파일 시스템에 접근하는 경로를 나타내고 있다. 사용자 프로세스는 VFS 하부의 파일 시스템이 어떤 것이든 상관없이 오직 VFS 계층에 정의되어 있는 공통 파일 모델을 통해서 파일 시스템의 자료를 제어할 수 있다. 즉 uJFFS처럼 사용자 프로세스로 동작하는 파일 시스템도 VFS를 거쳐서 접근하기 때문에 사용자는 일반 파일 시스템에 접근하는 것과 동일한 방법으로 엑세스할 수 있다.



(그림 1) 프로세스와 VFS 객체 사이의 상호동작

2.3 리눅스 기반의 플래시 파일 시스템

리눅스에서 동작 하는 플래시 파일 시스템은 여러 가지가 있다. 대표적인 것이 JFFS, JFFS2, yaFFS 등이 있다.

JFFS/JFFS2는 현재 리눅스에서 가장 널리 사용되는 플래시 파일 시스템이다. JFFS는 저 전력 임베디드 시스템에서 플래시 메모리를 저장장치로 이용할 수 있도록 한 파일 시스템으로 데이터와 메타 데이터가 플래시 칩에 순차적으로 저장되는 선형 구조를 가지고 있다. 이것은 jffs_raw_inode 구조체로 관리되며, 하나의 노드와 연결되어 있다. JFFS2는 기존의 플래시 파일 시스템이 플래시 메모리에 데이터에 접근하기 위해 일반 하드 디스크처럼 에뮬레이트 하는 방법을 사용하는 것에 비해 JFFS2는 플래시 메모리에 파일 시스템을 직접 사상한다[3]. 처음에는 NOR 플래시 메모리만을 지원하나, 최근에는 리눅스에서 NAND 플래시 메모리도 지원하고 있다. 그러나 JFFS/JFFS2 파일 시스템은 플래시 노드를 관리하기 위해서 많은 양의 메모리를 필요로 하며, 저널링 노드 검색과 파일 구조를 결정하기 위해 시스템 부팅 시에 상대적으로 많은 시간을 필요로 한다.

yaFFS(yet another Flash File System)는 스마트 미디어 또는 NAND 플래시 메모리에 최적화된 플래시 파일 시스템으로 앞서 지적한 JFFS/JFFS2 파일 시스템의 단점을 보완하여 RAM의 부하를 줄이며, 시스템 부팅 시간을 단축시킨다[14]. yaFFS2는 2KB 페이지를 지원하는 새로운 NAND 플래시를 지원하며, yaFFS보다 수행 속도를 향상 시켰다. 그러나 yaFFS는 아직까지 상용 제품을 위한 파일 시스템으로 도입하기에는 불안정하여 위험 부담이 존재하며, 기존 JFFS 파일 시스템처럼 커널 영역에서 수행되므로 파일 시스템의 오류가 커널 전체에 영향을 미칠 수 있다.

FTL(Flash Translation Layer)은 Intel에서 개발된 일종의 플래시 디바이스 드라이버로 DOS에서 사용하는 FAT 파일 시스템과 플래시 메모리 사이에 존재하는 계층으로 나타낼 수 있다. 플래시 메모리를 하드 디스크와 같은 블록 디바이스처럼 에뮬레이트 하여 운영체제 또는 파일 시스템이 플래시 메모리를 일종의 가상 블록 디바이스처럼 취급할 수 있게 한다[2].

3. uJFFS의 특징

uJFFS 파일 시스템이 사용자 프로세스로 구현됨으로써 가지는 주요 특징은 다음과 같다.

- 파일 시스템이 커널에 포함되지 않기 때문에 커널의 크기를 줄일 수 있다. 따라서 커널은 더 단순한 구조를 가질 수 있으며, 보다 제한된 자원을 가진 임베디드 시스템에 적용하기가 쉬워진다.
- 파일 시스템이나, 디바이스 드라이버의 오류로 인한 시

스템의 고장확률이 낮아진다. 파일 시스템은 사용자 프로세스중의 하나로 동작하게 되므로, 오류를 발생시키더라도, 커널 자체에는 영향을 미치지 않으며, 안정된 시스템의 구성이 가능해진다.

- 파일 시스템이나 디바이스 드라이버가 다양한 사용자 API(Application Programming Interface)를 이용할 수 있어 개발이 용이해진다. 리눅스 커널에서는 지원하지 않는 라이브러리를 사용할 수 있어 개발이 용이해지며, 일반 응용 프로그램처럼 디버깅이 가능하여 개발 시간을 단축할 수 있다.

uJFFS는 사용자 영역에서 일반적인 사용자 프로세스로 동작하기 때문에 파일 시스템이 필요 없는 경우에는 프로세스를 종료할 수 있으며, 커널 내부에서 모듈로 동작하는 일반 파일 시스템 보다 더 유연한 구조를 가질 수 있다. 따라서 플래시 메모리가 시스템에서 분리되는 경우 커널은 파일 시스템과 관련된 정보를 유지할 필요가 없어 보다 단순한 커널을 구성할 수 있다.

리눅스 디바이스 드라이버를 커널과 독립적으로 개발하여 모듈의 형태로 커널에 적재할 수 있으나, 모듈이 동작하기 위해 커널에 적재된다는 것은 실제로는 커널의 일부로 존재하며 커널 코드로 동작하는 것을 의미한다[8]. 그러므로 커널에 의존적인 부분이 많으며, 필요에 따라 어셈블리로 작성된 코드는 커널에서 완전히 분리하여, 사용자 프로세스로 구성하는 것은 어렵다.

만일, 파일시스템에서 예기치 못한 오류가 발생하였을 때, 커널 내부 또는 모듈로 동작하는 파일시스템의 경우는 사용자가 그 오류에 대한 처리를 하지 못한 채로 전체 시스템의 운용이 중단되는 결과가 발생할 수 있다. 그러나 uJFFS 파일시스템은 사용자 프로세스로 단지 하나의 사용자 프로세스의 오류로써 발생하는 것이므로 전체 시스템이 중단되는 일이 발생하지 않으며, 다른 프로세스에 영향을 주지 않고 오류를 처리할 수 있다.

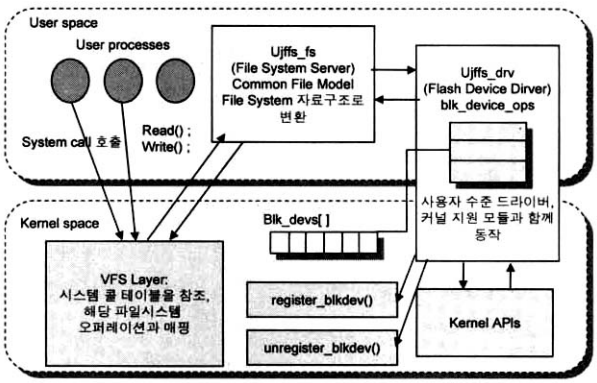
uJFFS는 플래시 디바이스 드라이버가 파일 시스템과 함께 사용자 프로세스로 동작하지만 일부 파일 시스템과 관련된 일부 커널 코드를 수행하기 위해 KSM(커널 지원 모듈, Kernel Support Module)을 이용한다. KSM은 일반적인 리눅스 디바이스 드라이버의 형태로 커널에 적재할 수 있으며, 사용자 프로세스로 동작하는 플래시 드라이버가 처리하지 못하는 일부 커널 함수를 실행하게 된다. 하지만 KSM은 커널 영역과 사용자 영역으로 나누어진 리눅스 커널에서 장치에 접근하기 위한 경로 제공의 역할을 수행하는 것이며, uJFFS는 파일 시스템의 자료구조가 커널영역의 메모리 공간에 생성되는 것이 아니라 사용자 영역의 메모리 자원을 사용하게 되며, 자료 구조에 대한 접근과 제어는 사용자 프로세스로 존재하는 uJFFS에 의해 제어된다.

4. uJFFS 설계 및 구현

4.1 파일 시스템의 구조

uJFFS는 플래시 파일 시스템과 플래시 디바이스 드라이버로 구성된다. (그림 2)는 uJFFS의 전체 구조를 나타낸 것으로 파일 시스템을 담당하는 ujffs_fs와 플래시 메모리는 제어하는 ujffs_drv, 그리고 사용자 프로세스로 동작하는 uJFFS에게 커널 서비스를 지원하기 위한 KSM으로 나누어진다. ujffs_fs와 ujffs_drv는 사용자 영역의 프로세스로 존재하면서 파일 시스템 서비스를 위한 서버의 형태로 동작한다. uJFFS는 파일 시스템과 플래시 디바이스 드라이버가 독립적인 프로세스로 구성이 되며 프로세스 간 통신을 통해서 필요한 자료를 주고받는다.

파일 시스템 서비스를 필요로 하는 응용 프로그램 프로세스는 VFS를 통해서 커널의 서비스를 받기 때문에 VFS 계층 아래에서 동작하는 파일 시스템이나 디바이스 드라이버의 수행 과정은 알 수 없다. 따라서 파일 시스템의 자료 전달 과정이나 동작은 사용자의 명령이나, 응용 프로그램의 파일 조작에 투명하게 구현될 수 있다.



(그림 2) uJFFS의 전체 구조

파일 시스템을 담당하는 ujffs_fs는 <표 1>에 서술한 함수에 의해 초기화 된다. 사용자가 파일 시스템을 사용하기 전, 즉 마운트되기 전에 프로세스를 시작해야 하며, ujffs_fs_attach 함수는 ujffs_fs 프로세스를 커널과 연결하는 역할을 한다. 프로세스와 관련된 정보를 유지하면서 커널과 자료교환에 필요한 정보를 초기화 하고, 커널 영역과 자료 교환에 이용되는 파이프를 생성한다. 이후 ujffs_register 함수가 호출되면, 먼저 플래시 메모리로부터 raw 데이터를 읽어서 VFS 자료구조로 변환하여 파일 시스템 트리를 생성하며, 슈퍼 블록 데이터를 읽어서 VFS에 파일 시스템을 등록한다. 파일 시스템에 사용되는 자료구조는 모두 사용자 메모리 공간에서 생성되며, 파일 오퍼레이션에 사용되는 함수 중 반드시 커널 영역에서 수행되어야 할 함수는 차후 4.2절에서 설명할 커널과의 통신에 의해서 필요한 자료를 전송받

는다. ujffs_fs_detach 함수는 파일 시스템이 언 마운트(unmount) 된 후 프로세스를 종료할 때 사용되며, 프로세스 정보들을 모두 제거한 후 파일 시스템을 유지하기 위해 할당되었던 메모리를 해제한다.

ujffs_drv 역시 사용자 프로세스로 실행된다. 드라이버의 등록은 uJFFS 라이브러리 함수를 통해 이루어지며, 플래시 디바이스 드라이버는 사용자 프로세스와 커널 인터페이스를 사용하기 위해 존재하는 커널 모듈인 KSM으로 구현된다. 드라이버 프로세스는 먼저 디바이스와의 연결을 위해 ujffs_drv_attach 함수를 호출하며, 리눅스 커널 내부에서 디바이스를 등록하기 위해 사용되는 커널 API인 register_blkdev 함수와 동일한 역할을 한다. 드라이버 프로세스의 등록 과정은 사용자 프로세스를 인식하고 I/O를 수행할 수 있도록 하기 위해 파일 기술자(file descriptor)와 디바이스 파일의 이름, 그리고 디바이스의 마이너 번호를 등록한다. 이와는 반대로 드라이버가 해제될 때는 ujffs_drv_detach 함수를 호출하여, 드라이버 프로세스를 장치에서 분리한다. <표 2>는 ujffs_drv의 초기화 함수이다.

〈표 1〉 ujffs_fs 초기화 함수

함 수 명	기 능
ujffs_fs_register	ujffs_fs서비스가 시작되면 파일 시스템의 슈퍼블록을 VFS에 등록하고 자료구조를 초기화 한다. 이후 서비스 요청이 오기를 기다린다.
ujffs_fs_attach	ujffs_fs가 KSM에 처음 연결될 때 호출된다. 파일 오퍼레이션과 관련된 자료구조를 초기화 한다.
ujffs_fs_detach	unmount후 ujffs_fs가 종료할 때 VFS에서 파일 시스템을 해제한다.

〈표 2〉 ujffs_drv 초기화 함수

구 분	함 수 명	기 능
ujffs_drv 프로세스	ujffs_drv_start	드라이버 프로세스의 서비스를 시작한다. 커널 모듈로부터 요청을 받아서 커널 모듈에 정의된 함수와 대응하는 오퍼레이션을 호출한다.
	ujffs_drv_attach	플래시 드라이버가 커널 모듈에 처음 연결될 때 호출된다. 연결되는 디바이스 파일 이름과, ioctl 정보, 디바이스 오퍼레이션 구조체의 포인터와, 드라이버 프로세스의 정보를 저장하고 있는 attach_record 구조체를 인수로 전달한다. 드라이버 프로세스의 id를 반환한다.
	ujffs_drv_detach	드라이버 프로세스가 커널 모듈에서 제거될 때 호출되며, 자료교환을 위해 생성한 파이프를 모두 해제한다.
KSM	attach_device	플래시 드라이버를 커널 모듈과 연결한다. 드라이버의 ioctl 오퍼레이션이 수행될 때 사용자 영역에서 attach_record의 정보를 복사해 온다. 플래시 드라이버의 정보와 마이너 번호로 드라이버를 식별하며, 드라이버 프로세스의 id를 반환한다.
	detach_device	커널 모듈에서 플래시 드라이버를 해제할 때 호출된다.

KSM은 플래시 드라이버로 전달되는 모든 요청을 파이프를 이용해서 ujffs_drv로 보낸다. 사용자 수준에서 요청이 처리가 되면, 드라이버는 다시 파이프를 이용해서 커널 모듈로 응답을 보낸다. 리눅스에서는 root 권한의 프로세스는 iopl과 ioperm 시스템 콜을 사용하여 I/O 포트를 직접 제어할 수 있다. 디바이스 드라이버에서는 다수의 사용자 프로세스가 동일한 디바이스에 접근하는 경우가 발생할 수 있다. 이 경우 KSM은 이전의 요청에 대한 응답을 기다리지 않고, 또 다른 요청을 사용자 드라이버로 전송한다. 이 경우 수행이 완료된 오퍼레이션은 진행 중인 오퍼레이션이 종료되기를 기다릴 필요가 없다. 각각의 요청은 식별 가능한 id를 가지고 있으므로 독립적으로 수행이 가능하다. 드라이버 프로세스의 등록과 해제는 모두 커널 모듈에서 ioctl 시스템 콜에 의해 처리된다. 초기화 작업 후 드라이버 프로세스는 사용자 영역에서 실행되며, 다른 사용자 프로세스에서 호출된 시스템 콜을 기다린다. 사용자 프로세스에서 호출된 시스템 콜은 최종적으로 (그림 3)에 정의된 오퍼레이션 구조체의 함수 포인터에 의해 처리된다.

```
/* flash device operation structure in driver process */
struct ujffs_operations uops = {
        ...
        fs_open,
        fs_release,
        fs_ioctl,
        fs_check_media_change,
        ...
        fs_revalidate,
        fs_message,
        fs_request
};
```

(그림 3) ujffs_drv 오퍼레이션 구조체

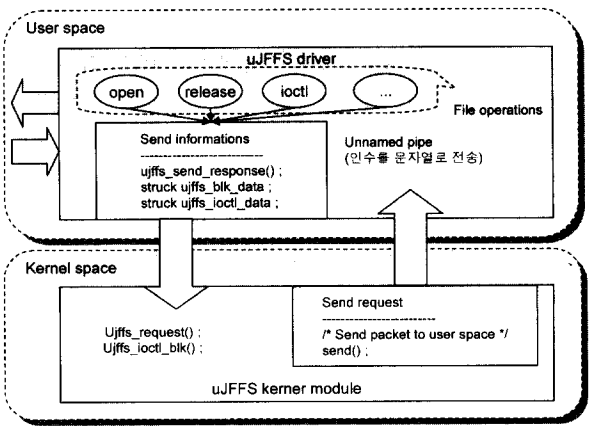
4.2 uJFFS와 커널의 통신

ujffs_drv에서 사용하는 자료구조는 일반적인 리눅스 디바이스 드라이버에서 사용하는 자료구조와 유사하지만, 사용자 영역에 존재하기 때문에, 커널 영역에 디바이스 드라이버의 존재를 알리고, 그 정보를 제공하기 위해 몇 가지 정보들이 추가된다. KSM에서는 사용자 영역의 드라이버 프로세스에 대한 정보를 driver_info 구조체에 저장하며, 커널 모듈이 다수 개 존재할 수 있는 드라이버 프로세스를 식별하는데 사용된다. driver_info 구조체는 플래시 디바이스의 이름과 파일 오퍼레이션의 포인터, 드라이버 프로세스의 id, ioctl함수 포인터 등의 정보를 가지고 있다.

uJFFS는 커널에 독립적인 사용자 프로세스로 동작하기 때문에 커널과의 자료 교환을 위한 통신이 필요하다. 커널과 uJFFS간의 통신은 크게 두 부분으로 나누어진다. uJFFS와 VFS계층, 그리고 uJFFS와 플래시 디바이스 드라이버 간의 통신이 그것이다.

uJFFS는 사용자 영역에 존재하므로 기존의 리눅스 시스템 콜의 실행 흐름으로는 사용자 영역에 있는 파일 시스템에 접근할 수 없다. 따라서 파일 시스템을 마운트할 때도 VFS가 인식할 수 없기 때문에 VFS 계층에서 sys_mount 시스템 콜에서 인수로 넘겨받은 파일 시스템의 이름이 "ujffs"일 경우, uJFFS 파일 시스템의 슈퍼블록을 읽어오기 위해서 KSM을 통해 사용자 영역으로 접근해서 uJFFS를 인식한 후 uJFFS의 등록을 수행할 수 있도록 시스템 콜의 실행 경로를 수정한다. 이후 VFS는 do_mount 함수를 호출하여 파일 시스템의 슈퍼블록과 디바이스 번호, 디바이스 파일 이름, 마운트 지점, 파일 시스템 형식, 마운트 플래그 등을 인수로 넘겨받아 파일 시스템을 마운트한다.

(그림 4)는 커널 모듈과 uJFFS간의 자료전달 방법을 나타낸 것이다. 이들 간의 통신은 두 개의 단 방향 파이프를 통해 이루어진다. 응용 프로그램에서 VFS 계층을 통해 KSM으로 전달된 모든 서비스 요청은 문자열 형태의 하나의 메시지로 변환되어 파이프를 이용해서 사용자 영역에 있는 플래시 드라이버로 전달된다. 플래시 드라이버에서 요청이 수행되면, 다른 파이프를 사용해서 결과를 KSM으로 반환한다. 파이프를 통해 전달되는 자료는 디바이스에 대한 정보와 그에 대한 ioctl 함수를 호출하기 위한 인수로 이들은 단일 문자열로 변환된 후 전송된다. 수신 측에서는 파이프를 통해 전달받은 문자열을 파싱(parsing)하여 요청에 해당하는 함수를 호출한다. 통신에 사용되는 문자열은 파일 시스템에서 제공하는 각각의 오퍼레이션 마다 전달해야하는 인수들이 다르므로 파일 오퍼레이션 마다 고유의 메시지를 생성하는 함수들에 의해 수행된다. 이 함수는 전송할 메시지의 종류, request id, 메시지를 받을 드라이버 프로세스에 관한 정보와 실제 전달할 데이터를 인수로 사용한다. 사용자 프로세스인 uJFFS와 커널 영역에서 동작하는 KSM사이의 자료교환은 리눅스 커널에서 사용자 프로세스의 정보를 얻기 위해 통상적으로 사용하는 커널 API인 copy_from_user와 copy_to_user를 사용하여 구현하였다.



(그림 4) 커널과 uJFFS간의 자료 전달

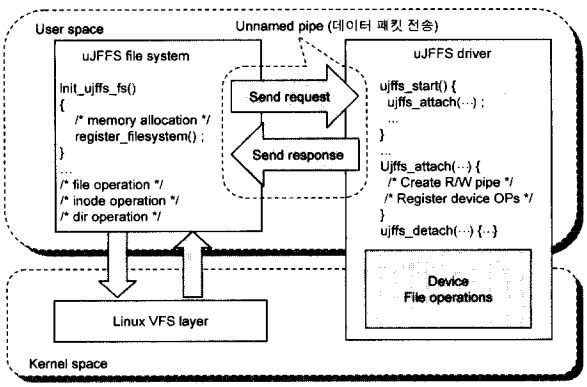
<표 3>은 ujffs_drv와 KSM 사이의 통신에 사용되는 인터페이스이다.

〈표 3〉 ujffs_drv와 KSM 간 통신 인터페이스

구 분	함 수 명	기 능
드라이버 프로세스	ujffs_send_response	이 함수는 커널 모듈에서 요청한 오퍼레이션의 수행 결과를 드라이버 프로세스 식별자와 요청 id와 함께 커널 모듈로 전달한다.
	ujffs_request_irq	커널 모듈에서 인터럽트가 발생할 때 해당 시그널로 드라이버 프로세스에 인터럽트의 발생을 알리기 위해서 호출된다.
KSM	send	driver_info 구조체를 참조하여 드라이버 프로세스에 패킷을 보낸다.
	receive	드라이버 프로세스에서 오퍼레이션의 수행 결과를 받아서 반환한다. 수신된 패킷은 요청 id에 따라 구분된다.

4.3 파일 시스템과 플래시 드라이버의 자료 교환

시스템 콜에 의해 VFS 계층에서 파일 시스템에 전달된 수행의 흐름은 최종적으로 디바이스 드라이버에 의해 수행되어야 한다. uJFFS는 내부에서 전통적인 리눅스 커널에 의한 시스템 콜 처리와는 다른 방법을 사용한다. (그림 5)는 uJFFS의 내부 통신을 나타낸 것이다. uJFFS 플래시 파일 시스템은 응용 프로그램에서 요청한 오퍼레이션을 수행하기 위해 최종적으로 플래시 드라이버 오퍼레이션을 수행해야 한다. ujffs_fs에서 ujffs_drv 프로세스의 오퍼레이션을 호출하는 방법은 <표 3>에 서술한 send 함수를 사용하여, KSM으로 패킷을 전달하는 방법을 사용한다. 즉 ujffs_fs와 KSM 사이에 두 개의 단 방향 파이프를 생성하고 이를 사용한다. 즉 ujffs_fs에서는 <표 3>에서 설명한 ujffs_send_response 함수를 사용하여 KSM에 디바이스 드라이버의 실행에 필요한 인수를 전달하며, 이것은 드라이버 프로세스와 커널 모듈간의 통신 방법과 동일하다.



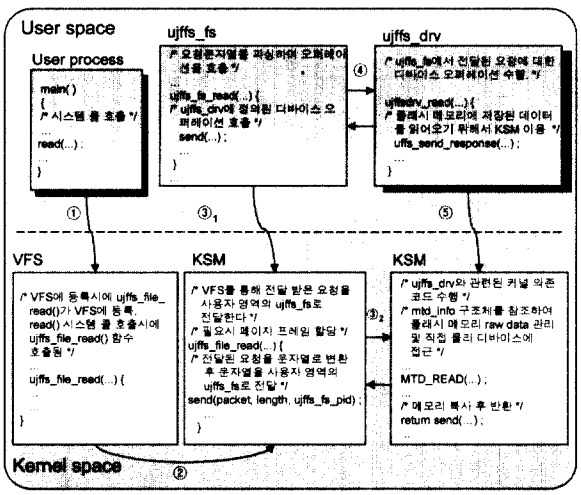
(그림 5) uJFFS의 내부 자료전달

많은 입출력 장치가 인터럽트 구동 방식으로 동작한다. 그러나 사용자 영역에서는 하드웨어 인터럽트를 직접 제어

하는 것이 불가능하므로 인터럽트 발생을 사용자 영역으로 전달 할 수 있는 기구가 필요하다. uJFFS에서는 하드웨어 인터럽트와 가장 유사한 시그널을 사용한다. ujffs_drv는 ioctl 시스템 콜을 사용해서 커널 모듈에 인터럽트 번호와 해당 인터럽트가 발생할 때 받아야 하는 시그널 번호를 요청한다. 커널 모듈에서는 ujffs_interrupt 함수에서 인터럽트가 발생할 때, 인터럽트 번호와, 드라이버 프로세스 정보, 인터럽트 발생 시 레지스터 상태정보를 인수로 받아서 사용자 드라이버에 전달한다. ujffs_drv는 특정 하드웨어 인터럽트가 발생했을 때 수행되어야 하는 인터럽트 핸들러를 연결하고 인터럽트 핸들러는 사용자 드라이버가 인터럽트를 처리할 수 있도록 시그널을 사용자 드라이버로 전달한다. 사용자 영역에서는 인터럽트를 직접 처리하는 것이 아니라 해당 시그널을 받아서 시그널 핸들러를 수행한다.

4.4 uJFFS상의 데이터의 접근

uJFFS의 동작기구는 4.2절과 4.3절에서 밝힌 바와 같다. 즉, JFFS를 사용자 영역에서 동작할 수 있도록 구현하였으며, 이를 위해 필요한 커널과 사용자 프로세스의 자료 전달 방법과 커널 지원을 설명하였다. 본 절에서는 임의의 쉘 명령 또는 사용자 프로세스가 uJFFS에 저장된 파일에 접근하는 과정을 설명한다.



(그림 6) read() 호출시의 실행 경로

(그림 6)은 사용자 프로세스가 read() 시스템 콜을 호출하였을 때 uJFFS로 접근하는 경로를 순서대로 서술한 것이다.

① 리눅스에서 파일 시스템 서비스를 요청하는 쉘 명령이나 사용자 프로세스는 파일 시스템과 관련된 시스템 콜을 호출한다. 사용자 프로세스가 커널 영역으로 접근할 수 있는 유일한 방법은 커널 시스템 콜을 호출하는 방법뿐이다[7]. uJFFS 파일 시스템에 접근하는 요청이 처리되는 과정을 read 시스템 콜을 예로 들면, 먼저 사용자 프로세스에 의해 사용자 라이브러리에서 시스템 콜 처리를 위해 0×80 인터럽트 발생 후 실행의 제어권이 커널 영역으로 넘어간다.

② 시스템 콜 테이블을 참조한 후 사용자 영역에서 요청한 시스템 콜 함수를 VFS 계층으로 전달하여 요청한 파일 시스템의 파일 오퍼레이션 중 호출된 시스템 콜에 사상된 함수를 호출한 후 최종적으로 디바이스 드라이버의 오퍼레이션을 수행한 후 결과를 반환하게 된다. VFS 계층까지는 인터럽트 핸들러와 시스템 콜 테이블을 거치는 등 실행 경로가 일반 파일 시스템과 동일하다. VFS에서는 uJFFS에 접근하는 경우에 이미 파일 시스템 마운트에서 등록한 "ujffs"라는 이름으로 파일 시스템을 식별한다. VFS에서는 파일을 read 시에는 uJFFS가 등록시에 정의된 ujffs_file_read 함수를 호출한다.

③$_1$ KSM에서는 실제 uJFFS 파일 시스템에 접근하게 되는데 ujffs_fs는 사용자 영역에 있으므로, 파일 시스템이 마운트 될 때 생성된 파이프를 통해 인수를 넘긴다. 이때, KSM은 ujffs_fs에게 사용자의 read 시스템 콜의 인수를 send 함수를 이용하여 문자열로 변환 후 전송하게 된다. 사용자 영역에서 동작하는 ujffs_fs_read 함수는 내부에서는 JFFS 파일 시스템과 기본적으로 동일한 동작을 수행한다. read할 데이터가 현재 페이지 캐시에 포함되어 있다면 실제 페이지를 읽기 위해서 사용자 프로세스에 요청하지 않으며, 만일 포함되어 있지 않다면 포함되어 있는 페이지를 얻기 위해서 페이지 프레임을 할당한다. 사용자 영역에서 지원하지 않는 커널 자료구조와 자료 형은 사용자 영역에서 다시 정의하여 사용하였으며, 동기화를 위한 뮤텍스나 세미포어는 사용자 C 라이브러리에 정의된 API를 사용하여 재 정의되었다.

③$_2$ 만일 읽기위한 페이지가 존재하지 않아서 페이지를 요청해야 하는 경우에는 현재 할당된 페이지를 실제 MTD 디바이스를 통해 얻어지는 raw data를 저장하기 위하여 사용해야 하므로 KSM에서 실제 raw data를 읽어들이는 모듈에게 페이지 정보를 넘겨준다. 이것은 모듈간 통신 방법(intermodule communication)을 사용한다.

④ ujffs_fs_read 함수는 KSM에서 전달받은 문자열을 파싱하여 ujffs_drv에게 read 요청을 넘긴 후 ujffs_drv는 내부에 정의된 ujffsdrv_read 오퍼레이션을 수행하게 된다. 이 단계에서 ujffs_drv는 직접 하드웨어를 제어하는 것이 불가능하므로, 최종적으로 물리 디바이스에 접근하기 위해서 KSM을 통해 플래시 메모리에 접근하여 데이터를 읽는다.

⑤ 여기서는 uJFFS와 연결되어 있는 플래시 메모리의 정보를 저장하고 있는 mtd_info 구조체의 내용을 참조하여 KSM이 실제 플래시 메모리에 접근할 수 있는 하드웨어 제어 코드로 구현된다. 이때 ③$_2$를 통하여 얻은 페

이지정보를 사용하여 이 페이지로 플래시 메모리의 데이터를 옮겨온다. 그 후 페이지 내의 필요한 데이터를 읽어 들여 데이터를 반환하게 된다. 이 시점에서 또 한 번의 문맥 교환이 발생한다. 플래시 메모리에서 읽어온 데이터를 반환할 때 역시 앞서 설명한 경로를 거슬러 올라가게 된다. 이후 VFS에서는 반환된 데이터를 페이지 프레임에 채운 후 요청된 바이트 수만큼의 데이터를 사용자 프로세스의 주소 공간으로 복사한다.

프로세스간의 자료 교환 또는 ujffs 프로세스와 KSM간의 자료교환은 모두 문자열 값을 이용한 파싱하는 방법을 사용하는데, read 오퍼레이션을 수행 후 반환되는 과정에는 이 문자열에 데이터 값이 포함되게 된다. 반대로 write 시스템 콜을 호출하였다면, 플래시 메모리에 기록해야하는 값 프로세스 간에 전달되는 인수와 함께 문자열로 변환되어 전송된다.

5. 성능 평가

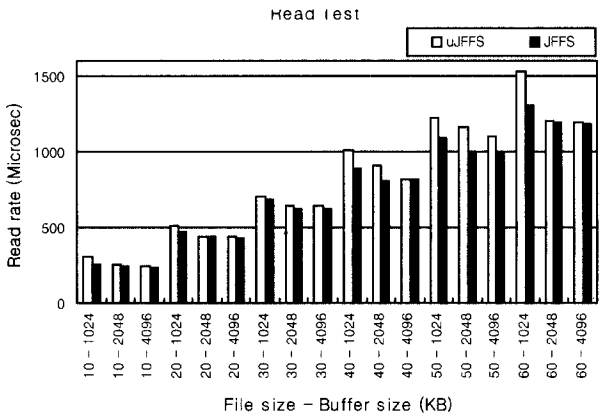
본 장에서는 앞서 구현한 uJFFS의 읽기, 쓰기에 소요되는 시간을 측정하여 성능을 시험한다. uJFFS와의 비교 측정에 사용하는 파일 시스템은 기존 JFFS 파일 시스템을 그대로 사용한다. 본 실험에서는 읽기/쓰기 속도만을 측정하며, 저널링 기능에 대한 성능 평가는 실시하지 않는다. uJFFS는 파일 시스템뿐만 아니라, 디바이스 드라이버까지 포함하고 있으며, JFFS는 리눅스 커널에 포함된 MTD(Memory Technology Device) 드라이버를 사용하므로, 본 실험에서는 순수한 파일 시스템뿐만 아니라, 디바이스 드라이버의 동작 속도까지도 실험 결과에 포함된다.

실험 환경은 Intel StrongARM SA1110 CPU와 32MB의 플래시 메모리를 내장한 EZ Board-M01 시험보드에서 이루어졌으며, 리눅스 커널 2.4.18에 ARM용 패치와 StrongARM용 패치를 적용하여 사용하였다. 실험 항목은 각 파일 시스템의 읽기, 쓰기에 소요되는 시간을 측정하였다. 소요 시간 측정 방법은 사용자 프로세스에서 read()와 write() 시스템 콜을 호출한 후 사용자 프로세스로 반환될 때까지의 시간을 측정하는 방법을 사용하였다. 실험 방법은 10KB~60KB까지 파일의 크기를 10KB씩 증가시키면서 각각의 파일에 대해 1024Byte, 2048Byte, 4096Byte의 버퍼를 적용 시켜 읽기, 쓰기 속도를 측정하여 동작 속도를 측정하였다.

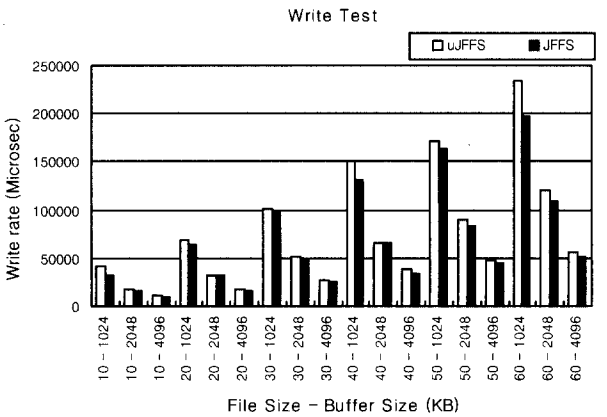
(그림 7)과 (그림 8)은 uJFFS와 JFFS의 성능 실험 결과이다. 각각의 실험은 모두 10회씩 반복 수행 후 그 평균값을 나타낸 것이다. 실험 결과, 예상했던 대로 uJFFS가 JFFS보다 시스템 콜 수행에 좀 더 시간이 소요되었음을 알 수 있다. 두 파일 시스템을 비교해보면 uJFFS가 JFFS보다 최고 1%, 최고 10%의 속도저하가 나타났다.

JFFS는 커널의 일부로 동작하기 때문에 하나의 사용자 요청을 처리하는 동안 사용자 프로세스로의 문맥 교환이 발생하지 않는다. JFFS는 물리 디바이스에 접근할 때까지의 서비스 요청 경로가 VFS에서 파일 시스템을 거쳐 디바이스 드라이버까지 자료의 이동시 모두 커널 영역에서 작업이 수행된다. 즉 사용자 프로세스의 입출력 요청을 처리하기 위해 커널 영역으로 진입하게 되면 플래시 드라이버에 의해 최종적으로 물리적인 액세스 후 결과 값을 해당 프로세스로 반환하기 전까지는 문맥 교환이 일어나지 않는다. 따라서 VFS와 디바이스 드라이버간의 자료 교환이 용이한 반면, uJFFS는 사용자 영역에서 동작하기 때문에, 커널 영역과 사용자 영역 사이의 통신에 더 많은 부하를 요구한다.

하나의 요청을 처리하는 과정 중 커널 스케줄러에 의해 다른 프로세스에게 CPU 점유권을 넘겨주는 경우도 발생하며 여러 개의 프로세스를 거치는 동안 요청을 문자열로 복사하여 전달하기 때문에 메모리 복사회수가 증가하여 JFFS보다는 속도가 상대적으로 느리게 된다. (그림 7)의 Read Test 그래프에서 파일사이즈가 60KB, 버퍼사이즈가 1KB일 때 결과치가 이러한 현상을 단적으로 보여주는 예이다.



(그림 7) uJFFS와 JFFS의 읽기 성능 평가 결과



(그림 8) uJFFS와 JFFS의 쓰기 성능 평가 결과

(그림 7)에 나타난 실험 결과는 파일의 크기와 버퍼의 크기에 따라 거의 규칙적인 비례 관계를 잘 나타내고 있다.

즉 커널 내부에서 페이지 단위로 데이터를 처리하므로, 쓰기 작업에 사용한 버퍼의 크기가 클 경우 페이지에 데이터를 빠르게 복사할 수 있으므로, 페이지 크기와 쓰기 속도는 비례하는 것을 그래프를 통해 확인 할 수 있다.

본 실험 결과, uJFFS는 설명한 바와 같이 기존 JFFS 파일 시스템보다 상대적으로 많은 문맥 교환과 자료의 이동 경로가 복잡하지만, 수 KB에서 수십 KB의 크기를 가진 파일의 읽기/쓰기 시간은 근소하게 느리거나 거의 대등하게 나타났다.

6. 결 론

본 연구에서 구현한 uJFFS는 기존 리눅스에서 커널 영역에서 동작하는 JFFS 플래시 파일 시스템을 사용자 영역에서 구현하여 파일 시스템을 커널에 독립적으로 구성하는데 그 목적을 두었다. 이는 마이크로 커널 구조에서 운영체제의 기본적인 기능은 커널 내부에 구성하고 디바이스 드라이버와 파일 시스템 등을 서버 프로세스로 구성하는 것처럼, 리눅스에서 uJFFS의 구현을 통해 대형 분산 시스템이나 범용 시스템을 위한 용도로 사용되는 마이크로 커널 구조가 소형 장비에 적용될 수 있음을 보여준다. 실제로 QNX와 같은 실시간 운영체제는 마이크로 커널 구조를 가지면서 임베디드 시스템에 적용되고 있다.

uJFFS는 파일 시스템과 이를 지원하기 위한 디바이스 드라이버 역시 사용자 영역에서 구현하였으며, 파일 시스템과 커널 자료구조의 사용을 최소화하면서 동작한다. 이는 리눅스 커널의 수정을 최소화하면서 마이크로 커널과 같은 사용자 수준의 파일 시스템 서비스와 디바이스 드라이버의 구현 가능성을 보여준다. 사용자 수준 운영체제 서비스는 기존 커널 영역 파일 시스템이나 디바이스 드라이버와 유사한 인터페이스를 제공하여 새로운 장치나 파일 시스템에 대한 개발 지원이 용이하고, 사용자에게는 투명하게 서비스할 수 있다는 장점이 있다. 또 사용자 영역에서 동작하게 되므로 파일 시스템이나, 디바이스 드라이버에서 발생하는 오류가 커널에 영향을 주지 않으므로, 시스템의 안정성은 더 증가한다.

사용자 수준에서 동작하는 uJFFS 파일 시스템은 커널 서비스가 아니기 때문에 시스템 부팅 시에는 마운트할 수 없다. 따라서 시스템 전체의 루트 파일 시스템으로 사용할 수는 없다. 또 커널과는 주소공간이 분리된 곳에 존재하기 때문에 사용자의 요청과 응답을 처리하는데, 문맥 교환 횟수가 보다 증가하여 앞서 실시한 실험 결과처럼 기존 플래시 파일 시스템 보다 실행 속도가 느리다는 단점이 있다. 성능 향상을 위해서 시스템 내에서 자료 이동 경로를 더 단순화할 필요가 있다. 현재는 기존 리눅스 파일 시스템의 접근과 동일하게 사용자 프로세스가 uJFFS에 접근하기 위해서는 일반 파일 시스템과 동일하게 VFS를 거쳐 해당 디바이스 드라이버를 통해 다시 사용자 영역에 있는 uJFFS로 접근해야한다는 경로상의 단점이 있다. 이를 개선하기 위해서는 uJFFS로 접근해야할 경우에는 KSM을 거치지 않고 직접 플래시 메모리에 접근할 수 있는 방법 등을 생각해볼 수 있다. 그러나 이러한 경우는 커널 영역을 거치지 않고 사용자 프로세스가 하드웨어에 직접 접근할 수 있는 방법을 제공해야하는 등의 커널 수정이 필요하다. 이외에 특정 응용 프로그램에 대해서 VFS 계층을 거치지 않고 직접적으로 파일 시스템에 접근하거나 디바이스에 접근할 수 있는 방법이 있을 수 있다. 커널 영역과 사용자 영역 사이의 잦은 자료 교환에 대한 응답성을 높이기 위해서는 uJFFS의 가장 큰 성능 저하 요인인 문맥 교환과 메모리 복사회수를 줄여야 한다. 이를 위해서 현재 ujffs_fs와 ujffs_drv로 나누어진 프로세스를 하나로 통합할 경우 두 프로세스간의 문맥 교환과 메모리 복사 횟수를 줄일 수 있다. 또 현재 ujffs_drv와 KSM 간 통신에서 주고받는 요청은 파이프를 통해 전달한 문자열 파싱에 의해 수행되고 있는데, 이를 함수 직접 호출이나 공유 메모리 사용 등의 방법으로 개선할 수 있는 연구가 필요하다.

참 고 문 헌

[1] Alessandro Forin, David B. Golub and Bershad, "An I/O System for Mach 3.0," Proceedings of the Usenix Mach Symposium, November, 1991.

[2] "Understanding the Flash Translation Layer(FTL) Specification," http://www.intel.com, December, 1998.

[3] David Woodhouse, "JFFS : The Journaling Flash File System," http://sources.redhat.com/jffs2, 2000.

[4] David B. Golub and ard P. Draves, "Moving the Default Memory Manager out of the Mach Kernel," Proceedings of the USENIX Mach Symposium, November, 1991.

[5] Hari Krishna Vemuri, "Userdev : A Framework For User Level Device Drivers In Linux," April, 2002.

[6] Randall W. Dean, "Data Movement in Kernelized Systems," Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures, pp.243-261, April, 1992.

[7] Daniel P. Bovet, Marco Cesati, "Understanding the LINUX KERNEL 2nd," O'reilly, December, 2002.

[8] Rubini, A., Corbet, J., "Linux Device Drivers 2nd," O'reilly, June, 2001.

[9] Brent B. Welch, John K. Ousterhout, "Pseudo Devices : User-Level Extensions to the Sprite File System," In Summer USENIX Conference, June, 1988.

[10] Fitzhardinge, J., "Userfs : A user file system for linux," ftp://www.tldp.org/HOWTO/Module-HOWTO/index.html, August, 2001.

[11] Fitzhardinge, J., "Userfs-Filesystems Implemented as User Processes," ftp://sunsite.unc.edu/pub/Linux/ALPHA/userfs, 1997.

[12] Jeremy Elson, "FUSD : A Linux Framework for User-Space Device," http://www.circlemud.org/jelson/software/fusd, Octber, 2001.

[13] Theodore Ts'o, "Standalone Device Drivers in Linux," Proceedings of the 1999 USENIX Annual Technical Conference, July, 1999.

[14] Charles Manning, "YAFFS(yet another Flash File System)," http://www.aleph1.co.kr.co.uk/armlinux/projects/yaffs, December, 2001.

권 우 일

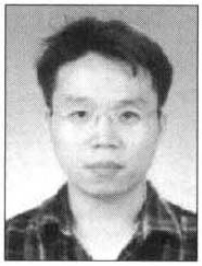
e-mail : wikwon@etri.re.kr

2000년 부경대학교 금속공학과(학사)

2003년 숭실대학교 대학원 컴퓨터학과 (석사)

2003년~현재 한국전자통신연구원 임베디드 S/W기술센터 연구원

관심분야 : 임베디드 리눅스, 실시간 시스템, 유비쿼터스 컴퓨팅



박 현 희

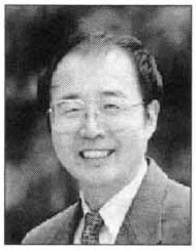
e-mail : darklight@realtime.ssu.ac.kr

2000년 숭실대학교 컴퓨터학부(학사)

2003년 숭실대학교 대학원 컴퓨터학과(석사)

2004년~현재 숭실대학교 대학원 컴퓨터학과 박사과정

관심분야 : 실시간 시스템, 임베디드 시스템, 리눅스 커널



양 승 민

e-mail : yang@computing.ssu.ac.kr

1978년 서울대학교 전자공학과(학사)

1983년 Univ. of South Florida 전자계산학 (석사)

1986년 Univ. of South Florida 전자계산학 (박사)

1986년~1987년 Univ. of South Florida 조교수

1988년~1993년 Univ. of Texas at Arlington 조교수

1996년~1998년 국회도서관 정보처리국장

1992년~현재 숭실대학교 컴퓨터학부 교수

관심분야 : 운영체제, 임베디드 시스템, 실시간 시스템, 결함 허용