9장. 파일 시스템 (File System)

이 장은 리눅스 커널이 지원하는 파일 시스템 안의 파일들을 어떻게 관리하는가를 설명한 다. 가상 파일 시스템(Virtual File System, VFS)과 리눅스 커널의 실제 파일 시스템이 어떻게 지원되는지를 설명한다.

리눅스의 가장 중요한 특징 중 하나는 많은 파일 시스템을 지원한다는 것이다. 이렇게 함으 로써 리눅스는 유연성을 갖게 되었고 다른 많은 운영체제와 잘 공존할 수 있게 되었다. 리 눅스를 처음 많들었을 때는 ext, ext2, xia, minix, umsdos, msdos, vfat, proc, smb, ncp, iso9660, sysv, hpfs, affs, ufs의 15가지 파일 시스템을 지원했고, 당연히 시간이 지남에 따라 더 많은 것이 추가되었다.

리눅스는 - 유닉스와 마찬가지로 - 시스템이 사용할 수 있는 각각의 파일 시스템이 장치 식 별자(드라이브 숫자나 이름)로 접근되는 것이 아니라 하나의 계층적인 트리 구조로 통합해 들어가서 파일 시스템이 마치 하나인 것처럼 보이게 한다. 리눅스는 이 하나의 파일 시스템 에 새롭게 마운트하는 파일 시스템을 덧붙인다. 모든 파일 시스템은 어떤 타입이든지 하나 의 디렉토리에 마운트되어, 마운트된 파일 시스템의 파일들이 그 디렉토리의 내용을 구성한 다. 이러한 디렉토리를 마운트 디렉토리 또는 마운트 포인트라고 부른다. 파일 시스템의 마 운트가 해제되면 마운트 디렉토리가 원래 가지고 있던 파일들이 다시 드러난다.

디스크가 초기화될 때 (가령, fdisk를 사용하여) 디스크는 파티션 구조를 가지게 되는데, 이 것은 물리적인 디스크를 논리적으로 몇 개의 파티션으로 분리하는 것이다. 각 파티션은 하 나의 파일 시스템 - 예를 들어, EXT2 파일 시스템 - 을 가지게 된다. 파일 시스템은 물리적 인 장치의 블럭에, 파일을 디렉토리나 소프트 링크 등과 함께 논리적인 계층 구조로 구성한 다. 파일 시스템을 담을 수 있는 장치는 블럭 장치이다. 시스템에 있는 첫번째 IDE 디스크 의 첫번째 파티션인 /dev/hda1은 블럭 장치이다. 리눅스 파일 시스템은 이러한 블럭 장치 들을 단순히 일렬로 늘어놓은 블럭의 모음으로 간주하며, 그 밑에 있는 물리적인 디스크에 대해서는 알지도 못하고 상관도 하지 않는다. 장치의 특정 블럭을 읽으라는 요구를 그 장치 에게 의미있는 요소들, 즉 특정 트랙, 섹터, 실린더 등 하드 디스크상에 그 블럭이 있는 위 치로 매핑하는 것은 각 블럭 디바이스 드라이버의 역할이다. 파일 시스템은 어떤 장치가 그 블럭을 가지고 있든지 간에 똑같은 방법으로 보고, 느끼고, 동작해야 한다. 게다가, 리눅스 파일 시스템을 사용하면, 이러한 다른 파일 시스템이 다른 하드웨어 컨트롤러에 의해 조작 되는 다른 물리적 매체에 있는 것은 전혀 문제가 되지 않는다 (적어도 시스템 사용자에게는 그렇다). 파일 시스템은 심지어 로컬 시스템에 있지 않을 수도 있다. 네트웍 연결로 원격지 에서 마운트된 디스크일 수도 있는 것이다. 리눅스 시스템이 SCSI 디스크에 루트 디렉토리 를 가지는 다음 예를 보자.

```
A       E       boot    etc     lib     opt     tmp     usr
C       F       cdrom   fd      proc    root    var     sbin
D       bin     dev     home    mnt     lost+found
```

파일을 가지고 작업하는 사용자도 프로그램도, /C가 사실은 시스템의 첫번째 IDE 디스크에 있는 VFAT 파일 시스템이 마운트된 디렉토리라는 것을 알 필요가 없다. 이 예에서 (사실은 필자의 집에 있는 리눅스 시스템이다), /E는 두번째 IDE 컨트롤러에 연결된 마스터 IDE 디 스크이다. 첫번째 IDE 컨트롤러는 PCI이며, 두번째 것은 IDE CDROM도 제어하는 ISA 컨트 롤러라는 것은 전혀 상관이 없다. 나는 모뎀과 PPP 네트웍 프로토콜을 사용해서 내가 일하 는 곳에 전화를 걸어, 내 알파 AXP 리눅스 시스템의 파일 시스템을 /mnt/remote에 원격 으로 마운트할 수도 있다.

파일 시스템의 파일들은 데이터의 집합이다. 이 장은 filesystems.tex라는 아스키 파일 에 들어 있다. 파일 시스템은 파일에 담긴 데이터 뿐만 아니라, 파일 시스템의 구조도 가지 고 있다. 파일 시스템의 구조에는 리눅스의 사용자나 프로세스가 볼 수 있는 파일, 디렉토리 에 대한 소프트 링크, 파일 보호 정보와 같은 것들이 포함된다. 이러한 정보들은 안전하고 신뢰성있게 저장되어야 하며, 따라서 운영체제의 기본적인 무결성은 그 파일 시스템에 달려 있다. 아무도 수시로 자료나 파일이 손상되는 운영체제를 사용하려 하지 않은 것이다[1].

리눅스가 처음 사용했던 미닉스(Minix)파일 시스템은 제한적이고 성능이 좋지 못했다. 파일 이름이 14자를 넘지 못했고 (그래도, 8.3 제한보다는 낫다), 파일 크기가 64M바이트로 제한 되었다. 64M바이트는 얼핏 보기에 충분할 것 같지만, 일반적인 데이터베이스를 저장하기 위 해서는 훨씬 큰 파일이 필요하다. 리눅스 전용으로 설계되었던 첫번째 파일 시스템은 확장 파일 시스템(Extended File System, EXT)으로, 1992년 4월에 소개되었고 많은 문제점을 해결했 지만 아직은 성능이 떨어졌다. 그래서, 1993년에 2차 확장 파일 시스템(Second Extended File System, EXT2)이 추가되었다. 이 장의 뒷부분에 자세히 설명될 파일 시스템이 바로 이것이 다.

리눅스에 EXT 파일 시스템이 추가되었을 때, 중요한 발전이 있었다. 실제 파일 시스템이 가 상 파일 시스템(Virtual File System, VFS)이라는 인터페이스 계층을 통해서 운영체제와 운영체 제의 서비스로부터 분리된 것이다. VFS 덕분에 리눅스는 VFS를 지원하는 서로 다른 많은 파일 시스템들을 사용할 수 있게 되었다. 리눅스 파일 시스템의 모든 세세한 것들이 소프트 웨어에 의해서 변환되어서, 모든 파일 시스템이 리눅스 커널의 다른 부분들과 그 위에서 실 행되는 프로그램에게는 같은 것으로 보인다. 또한, 리눅스의 가상 파일 시스템은 많은 다른 파일 시스템을 동시에 구별없이 마운트할 수 있게 해 준다.

리눅스 가상 파일 시스템은 그 안의 파일을 가장 빠르고 효율적으로 사용할 수 있도록 구현 되었다. 또한, 파일과 그 안의 자료가 정확하게 유지될 수 있어야만 한다. 이러한 두가지 요 구 조건은 서로 상반될 수 있다. 리눅스 VFS는 마운트되어 사용중인 각각의 파일 시스템의 정보를 메모리에 캐시한다. 파일이나 디렉토리가 생성, 삭제되거나 자료가 입력될 때 파일 시스템과 캐시 안의 자료를 정확하게 수정하기 위해서 많은 주의가 필요하다. 만약 실행중 인 커널안에서 파일 시스템의 자료구조들을 볼 수 있다면, 파일 시스템이 자료들을 읽거나 쓰는 것도 볼 수 있을 것이다. 접근하려는 파일이나 디렉토리를 나타내는 자료구조는 디바 이스 드라이버가 작업을 하거나, 자료를 꺼내거나 저장하는 과정에서 만들어지고 없어진다. 캐시 중에서 가장 중요한 것은 버퍼 캐시(Buffer Cache)인데 이것은 각각의 파일 시스템이 그 아래에 있는 블럭 장치에 접근하는 방법에 통합되어 있다. 블럭에 접근하면 그 블럭은 버퍼 캐시에 들어가고 상태에 따라 여러가지 큐에 들어 있게 된다. 버퍼 캐시는 데이터 버 퍼를 캐시할 뿐만 아니라, 블럭 디바이스 드라이버와의 비동기적인 인터페이스의 관리도 도 와준다.

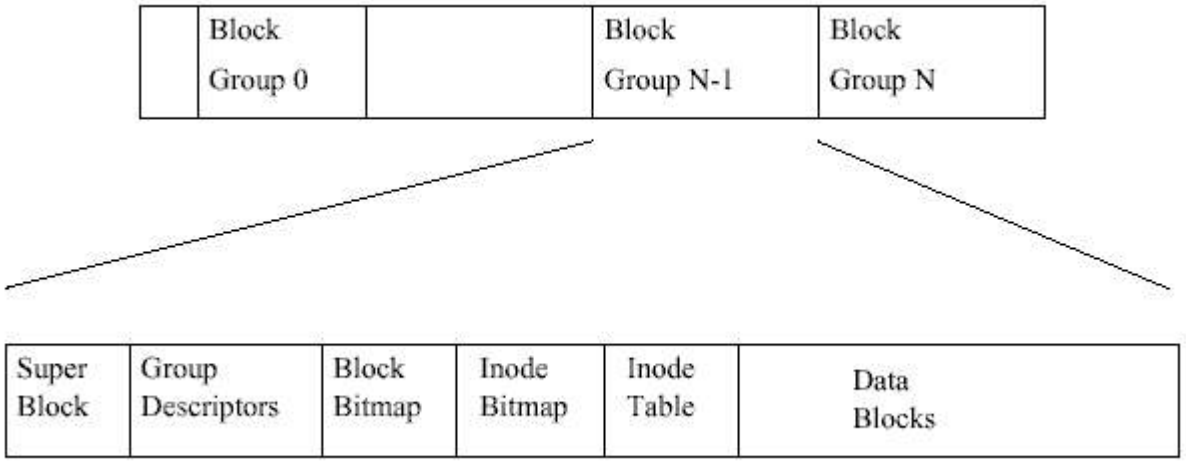

그림 9.1 : EXT2 파일시스템의 물리적 배치도

9.1 2차 확장 파일 시스템 (EXT2)

2차 확장 파일 시스템은 리눅스를 위한 확장성있고 강력한 파일 시스템으로 R?my Card가 고안한 것이다. 이는 현재까지 리눅스 공동체에서 만든 것 중 가장 성공적인 파일 시스템일 뿐만 아니라 현재 배포되고 있는 모든 리눅스 배포판의 기반을 이루고 있다. 다른 수많은 파일 시스템과 마찬가지로 EXT2 파일 시스템은 파일에 들어있는 데이터는 데이터 블럭에 저장된다는 것을 전제로 하고 있다. 모든 데이터 블럭의 크기는 같다. 물론 서로 다른 EXT2 파일 시스템에서는 크기가 다를 수 있다. 그리고 특정 EXT2 파일 시스템에서의 블럭의 크 기는 (mke2fs 명령을 통해) 파일 시스템이 만들어 질 때 결정된다. 모든 파일의 크기는 블 럭의 크기에 따라 올림이 된다. 만약 블럭의 크기가 1024바이트일 때, 크기가 1025바이트인 파일은 1024바이트 블럭 두개를 차지하게 된다. 이는 파일 하나당 평균 블럭의 절반 크기 만큼을 낭비하고 있다는 것을 의미한다. 대개 컴퓨터에서는 CPU의 메모리 사용량과 디스크 공간의 활용도 사이에 트레이드 오프(trade off)가 발생한다. 대부분의 운영체제와 마찬가지로 이러한 경우에 리눅스는 CPU의 부담을 줄이기 위하여 디스크의 활용도를 희생한다. 파일 시스템의 모든 블럭이 데이터만을 저장하는 것은 아니다. 어떤 블럭에는 파일 시스템의 구 조를 표현하는 정보를 담고 있어야 한다. EXT2는 파일 시스템 배치도를 정의하기 위하여 시 스템내의 각 파일을 inode 자료구조로 표현한다. inode는 파일의 데이터가 어느 블럭에 들어 있는지, 파일에 대한 접근 권한, 파일의 수정 시간, 파일의 종류 등을 나타낸다. EXT2 파일 시스템의 모든 파일은 각기 하나의 inode에 의하여 표현되며 각각의 inode는 각각을 구분할 수 있는 고유의 번호를 갖고 있다. 파일 시스템의 모든 inode는 inode 테이블에 들어 있다. EXT2의 디렉토리는 (그 자체도 inode로 표현되는) 단지 좀 별난 파일일 뿐이며 그 디렉토리 에 속하는 파일들의 inode에 대한 포인터를 갖고 있다.

그림 9.1은 EXT2 파일 시스템이 블럭 구조로 된 장치에서 블럭을 어떻게 차지하고 있는 지 배치 상태를 보여준다. 파일 시스템에 관한 한 블럭 장치는 그저 읽고 쓸 수 있는 일련의 블럭일 뿐이다. 파일 시스템은 실제 매체의 어느 곳에 블럭이 씌어야 하는지에 대해 신경 쓸 필요가 없다. 그것은 디바이스 드라이버가 알아서 할 일이다. 파일 시스템이 그 파일 시 스템을 담고 있는 블럭 장치로부터 정보나 데이터를 읽으려고 한다면 단지 해당 디바이스 드라이버에게 몇 개의 블럭을 읽어달라고 요청하기만 하면 된다. EXT2 파일 시스템은 자신 이 차지하고 있는 논리적인 파티션을 다시 블럭 그룹으로 쪼갠다. 각 블럭 그룹은 파일 시 스템에서 무결성의 핵심이 되는 정보를 중복해서 갖고 있으며, 실제 파일과 디렉토리를 정 보와 데이터의 블럭으로 갖고 있다. 이 중복은 파일 시스템이 깨지는 등의 재난이

발생해서 파일 시스템의 복구가 필요할 때 필수적이다. 다음 소단원에서 각 블럭 그룹의 내용을 더 자세히 설명한다.

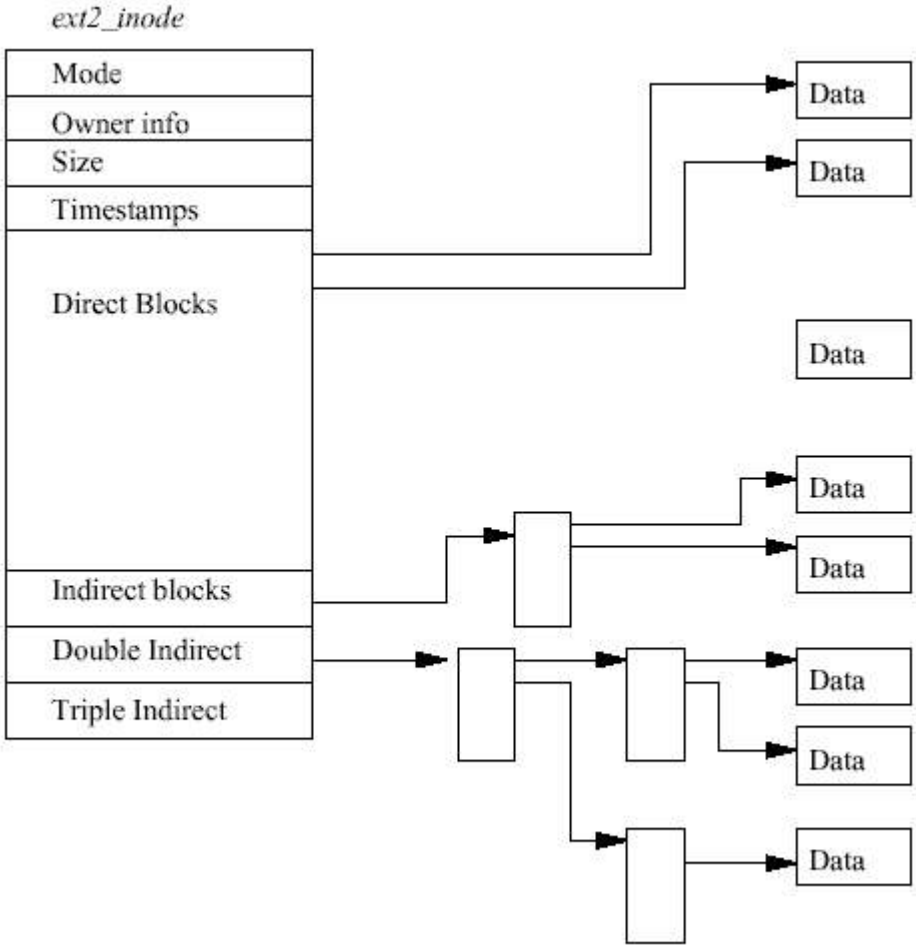

그림 9.2 : EXT2 inode

9.1.1 EXT2 inode

EXT2 파일 시스템에서 inode는 가장 기본이 되는 단위이다. 파일 시스템의 모든 파일이나 디렉토리는 각기 단 하나의 inode에 의하여 표현된다. 각 블럭 그룹을 위한 EXT2 inode는 어 떤 inode가 할당되었는지 아닌지를 추적하기 위한 비트맵과 함께 inode 테이블에 저장된다. 그림 9.2는 EXT2 inode의 형태를 보여준다. 저장되는 정보에는 다음과 같은 항목이 있다.

- **모드(mode)** 여기에는 이 inode가 어느 파일에 해당하는지를 나타내는 정보와, 접근권한을 나타내는 정보가 저장된다. EXT2에서 하나의 inode는 하나의 파일, 디렉토리, 심볼릭 링크, 블럭 장치, 문자 장치 또는 FIFO를 나타낸다.
- **소유자 정보(Owner Information)** 이 파일 또는 디렉토리에 대한 사용자와 그룹 식별자이다. 이 정보를 이용하여 파일 시스템은 접근권한을 제대로 관리할 수 있게 된다.
- **크기(Size)** 파일의 크기를 바이트 단위로 가지고 있다.

- **타임스탬프(Timestamps)** inode가 만들어진 시간과 최종적으로 수정된 시간을 기록한다.
- **데이터블럭(Datablocks)** 이 inode가 표현하고 있는 데이터가 저장된 블럭에 대한 포인터. 맨 앞의 열두개의 포인터는 이 inode가 표현하고 있는 데이터를 저장한 실제 블럭에 대한 포인터이며 마지막 세개의 포인터는 점점 더 높은 수준의 간접적인 연결을 갖고 있다. 예를 들어, 이중 간접 블럭 포인터(double indirect block pointer)는 데이터 블럭에 대한 포인 터들의 블럭에 대한 포인터들의 블럭을 가리키고 있다. 따라서, 길이가 12개 데이터 블럭 이하인 파일은 그 보다 큰 파일보다 훨씬 빨리 액세스 된다.

EXT2 inode는 특별 장치 파일을 표현할 수도 있다는 점에 주목하여야 한다. 이들 파일은 실 제 파일은 아니지만 장치를 액세스하는데 사용되는 프로그램을 다룬다. /dev 디렉토리 아래 의 모든 장치 파일은 프로그램이 리눅스 장치를 액세스할 수 있도록 하기 위하여 거기에 있 는 것이다. 예를 들어 마운트 프로그램은 마운트하려는 장치 파일을 인자로 사용한다.

9.1.2 EXT2 수퍼블럭(Superblock)

수퍼블럭에는 그 파일 시스템의 기본적인 크기나 모양에 대한 설명이 들어 있다. 여기에 들 어 있는 정보를 이용하여 파일 시스템 관리자는 파일 시스템을 활용하고 유지한다. 보통 파 일 시스템이 마운트 될 때에는 블럭 그룹 0에 들어 있는 수퍼블럭을 읽어들인다. 하지만, 모 든 블럭 그룹에는 똑같은 복사본이 있어서 파일 시스템이 깨지는 경우를 대비하고 있다. 여 기에 들어 있는 정보에는 다음과 같은 것들이 있다.

- **매직 넘버(Magic Number)** 이 값은 마운트하는 소프트웨어로 하여금 이것이 진짜 EXT2 파 일 시스템의 수퍼블럭이라는 것을 확인케한다. 현재 버전의 EXT2에서는 0xEF53으로 되 어 있다.
- **개정 레벨(Revision Level)** 이 값은 메이저 개정 레벨과 마이너 개정 레벨로 구성되며, 마운 트 프로그램이 어떤 특정한 버전에서만 지원되는 기능이 이 파일 시스템에서 지원되는지 아닌지를 확인하는데 사용된다. 또한 기능 호환성 항목라는 것이 있어서 마운트 프로그 램이 이 파일 시스템에서 안전하게 사용할 수 있는 기능이 무엇인지를 판단할 수 있도록 해준다.
- **마운트 횟수(Mount Count)와 최대 마운트 횟수(Maximum Mount Count)** 이 두개의 값을 이용하여 시스템은 파일 시스템 전부를 검사할 필요가 있는지를 확인할 수 있다. 마운트 횟수는 파일 시스템이 마운트될 때 마다 1씩 증가한다. 그리고 그 값이 최대 마운트 횟 수와 같아지면 "최대 마운트 횟수에 도달하였습니다, e2fsck를 실행하는 것이 좋습니다[2]" 라는 메시지가 표시된다.
- **블럭 그룹 번호(Block Group Number)** 현재 보고 있는 수퍼블럭 복제본을 갖고 있는 블럭 그룹의 번호.
- **블럭 크기(Block Size)** 이 파일 시스템의 블럭 크기를 바이트 단위로 (예를 들어, 1024 바이 트) 표시한다.
- **그룹당 블럭수(Blocks per Group)** 하나의 그룹에 속하는 블럭의 수. 블럭 크기와 마찬가지로 파일 시스템을 만들때 정해진다.
- **프리 블럭(Free Blocks)** 파일 시스템내의 프리 블럭의 수.

- **프리 Inode(Free Inode)** 파일 시스템내의 프리 inode의 수.
- **첫번째 Inode(First Inode)** 파일 시스템내의 첫번째 inode의 inode 번호. EXT2 루트 파일 시스템에서 첫번째 inode는 "/" 디렉토리에 대한 디렉토리 엔트리이다.

9.1.3 EXT2 그룹 기술자(Group Descriptor)

각 블럭 그룹은 자신을 기술하는 자료구조를 가지고 있다. 수퍼블럭과 마찬가지로 모든 블 럭 그룹을 위한 그룹 기술자는 각 블럭 그룹에 복제되어 파일 시스템이 파괴되는 경우를 대 비한다. 각 그룹 기술자는 다음과 같은 정보를 갖고 있다 :

- **블럭 비트맵(Blocks Bitmap)** 이 블럭 그룹에서 블럭의 할당 상태를 나타내는 비트맵으로서 블럭의 수 만큼 있다. 이것은 블럭을 할당하거나 해제할 때 사용된다.
- **Inode 비트맵(Inode Bitmap)** 이 블럭 그룹에서 inode의 할당 상태를 나타내는 비트맵으로서 블럭의 수 만큼 있다. 이것은 inode를 할당하거나 해제할 때 사용된다.
- **Inode 테이블(Inode Table)** 이 블럭 그룹의 inode 테이블의 시작 블럭으로서 블럭의 수 만큼 있다. 각 inode는 다음에 설명하는 EXT2 inode 자료구조에 의해 표현된다.
- **프리 블럭 갯수(Free Blocks Count), 프리 Inode 갯수(Free Inode Count), 사용된 디렉토리 갯수(Used Directory Count)**

그룹 기술자는 잇달아 나타나서 전체적으로는 하나의 그룹 기술자 테이블을 형성한다. 각 블럭 그룹에는 수퍼블럭 바로 뒤에 그룹 기술자 테이블 전체가 놓여있다. EXT2 파일 시스템 에서 실제로 사용되는 것은 (블럭 그룹 0에 있는) 첫번째 복사본 뿐이다. 다른 복사본들은, 수퍼블럭의 복사본들과 마찬가지로, 원본이 깨질 경우를 대비하고 있다.

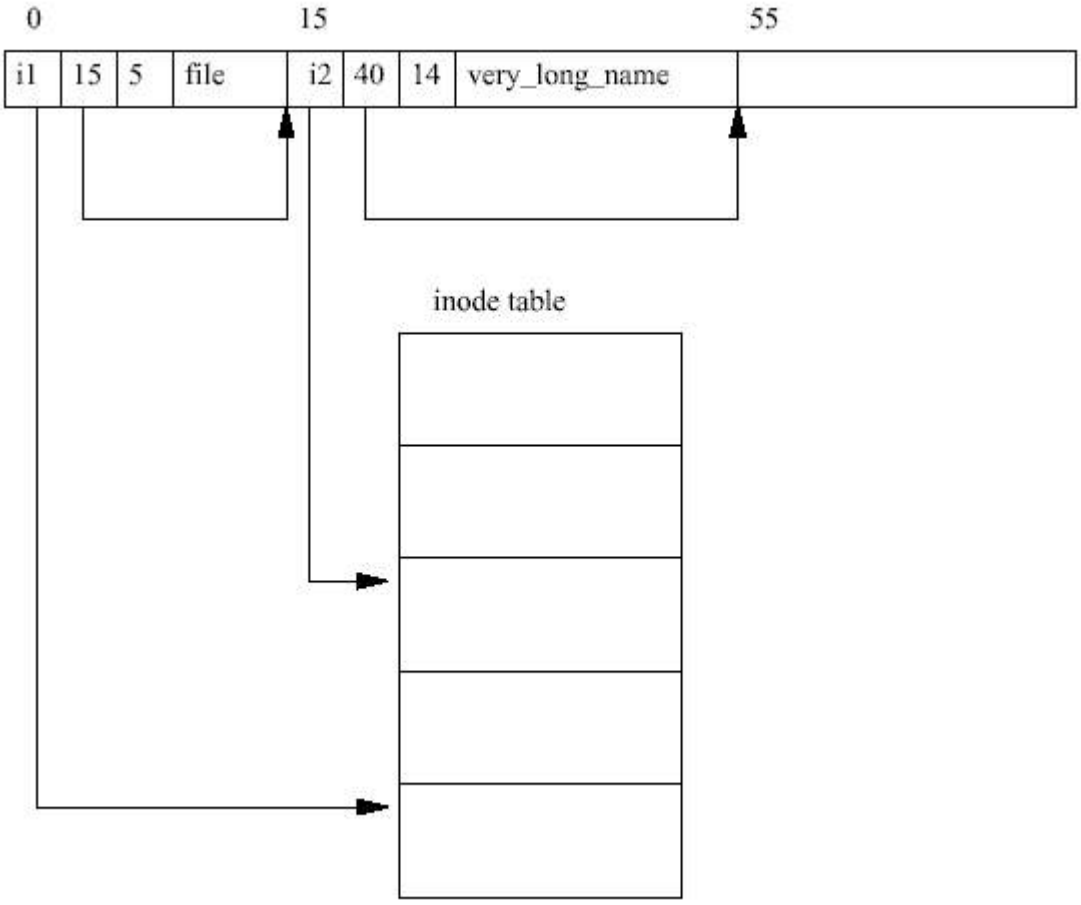

그림 9.3 : EXT2 디렉토리

9.1.4 EXT2 디렉토리

EXT2 파일 시스템에서 디렉토리는 파일 시스템내의 파일에 대한 접근 경로를 만들고 저장 하는 특별한 파일이다. 그림 9.3은 메모리 상에서의 디렉토리 엔트리의 모양을 보여준다. 디 렉토리 파일은 디렉토리 엔트리의 리스트이며 각각의 디렉토리 엔트리는 다음과 같은 정보 를 갖고 있다 :

- **inode** 이 디렉토리 엔트리에 해당하는 inode. 이 값은 블럭 그룹의 inode 테이블에 저장되어 있는 inode 배열에 대한 인덱스이다. 그림 9.3 에서 file이라는 이름의 파일에 대한 디 렉토리 엔트리는 i1이라는 번호의 inode를 참조하고 있다.
- **이름 길이(name length)** 이 디렉토리 엔트리의 길이를 바이트로 나타낸다.
- **이름(name)** 이 디렉토리 엔트리의 이름.

모든 디렉토리에서 처음 두 엔트리는 항상 "." 과 ".." 이다. 이는 각각 "현재 디렉토리" 와 " 부모 디렉토리" 를 의미한다.

9.1.5 EXT2 파일 시스템에서 파일 찾기

리눅스 파일 이름은 다른 유닉스의 파일 이름과 같은 형식으로 되어 있다. 이름은 앞에 슬 래시('/')가 붙은 디렉토리 이름이 이어지고 마지막에 파일 이름이 오는 형태이다. 예를 들어, 파일 이름이 /home/rusling/.cshrc이라면, /home과 /rusling은 디렉토리 이름이고 파 일 이름은 .cshrc이다. 다른 모든 유닉스 시스템과 마찬가지로 리눅스는 파일 이름 자체의 형식은 신경쓰지 않는다. 길이 제한도 없고, 인쇄 가능한 아무런 문자로 구성된다. 이 파일 을 나타내는 inode를 EXT2 파일 시스템 안에서 찾기 위해, 시스템은 파일 이름을 해석해서 한 디렉토리씩 처리하여 파일 자체를 얻게 된다.

처음 필요한 inode는 파일 시스템의 루트의 inode로, 그 inode 숫자값은 파일 시스템의 수퍼 블럭에서 얻는다. EXT2 inode를 읽기 위해서는 해당하는 블럭 그룹의 inode 테이블에서 찾아 야 한다. 예를 들어 루트 inode의 번호가 42라면 우리는 블럭 그룹 0의 inode 테이블의 42번 째 inode가 필요한 것이다. 루트 inode는 EXT2 디렉토리를 위한 것이다. 다시 말해서 루트 inode의 모드는 루트 inode가 디렉토리임을 나타내며 데이터 블럭에는 EXT2 디렉토리 엔트 리가 들어 있다.

home은 여러 디렉토리 엔트리 중의 하나일 뿐 이며 /home 디렉토리를 나타내는 inode의 번호를 알려준다. 이 디렉토리를 읽어서 (디렉토리를 읽으려면 우선 inode를 읽고 그 inode 가 가리키는 데이터 블럭으로부터 디렉토리 엔트리를 읽어야 한다.) rusling 엔트리를 찾 으면 그 엔트리는 /home/rusling 디렉토리를 나타내는 inode의 번호를 알려줄 것이다. 마 침내 우리는 /home/rusling 디렉토리를 나타내는 inode가 가리키는 디렉토리 엔트리를 읽어서 .cshrc 파일의 inode 번호를 찾은 다음, 이 번호를 이용하여 파일의 내용을 갖고 있는 데이터 블럭을 가져오게 된다.

9.1.6 EXT2 파일 시스템의 파일의 크기 변경

파일 시스템이 공통적으로 겪는 문제 중의 하나는 분할화 되는 경향이다. 파일의 데이터를 가지고 있는 블럭들은 파일 시스템 전체에 흩어지게 되고, 데이터 블럭이 더 멀리 떨어질수 록 한 파일의 데이터 블럭들을 순차적으로 접근하는 것이 점점 더 비효율적으로 된다. EXT2 파일 시스템은 이를 극복하려고 어떤 파일에 대한 새로운 블럭을 현재의 데이터 블럭들에 물리적으로 인접하도록 할당하거나 적어도 현재의 데이터 블럭과 같은 블럭 그룹에 할당하 려고 한다. 둘 다 실패했을 때만 다른 블럭 그룹에 있는 데이터 블럭을 할당한다.

프로세스가 파일에 데이터를 쓰려고 할 때마다, 리눅스 파일 시스템은 데이터가 파일에 마 지막으로 할당한 블럭을 넘어가는지 검사한다. 넘어간다면 이 파일을 위해 새로운 데이터 블럭을 할당해야 한다. 할당이 끝날 때까지 프로세스는 실행될 수 없다. 즉, 파일 시스템이 새로운 데이터 블럭을 할당하고 남은 데이터를 기록하도록 기다렸다가 계속 실행된다. EXT2 블럭 할당 루틴이 처음 하는 것은 이 파일 시스템에 대한 EXT2 수퍼블럭에 락을 거는 것이 다. 할당과 해제는 수퍼블럭에 있는 항목을 변경하며, 리눅스 파일 시스템은 둘 이상의 프로 세스가 동시에 변경하는 것을 허용하지 않는다. 다른 프로세스가 데이터 블럭을 할당하고자 하면 현재의 프로세스가 작업을 끝마치길 기다려야 한다. 수퍼블럭을 기다리는 프로세스는 정지되고, 수퍼블럭의 제어가 현재 사용자로부터 풀려날 때까지 실행되지 못한다. 수퍼블럭 의 사용은 온 순서에 따르며, 한 프로세스가 수퍼블럭의 제어권을 갖게 되면 작업을 종료할 때까지 제어를 갖고 있는다. 프로세스는 수퍼블럭에 락을 건 뒤 이 파일 시스템에 프리 블 럭이 충분히 남아있는지 확인한다. 만약 프리 블럭이 충분하지 않다면 더 이상 할당받으려 는 시도는 실패할 것이기 때문에 프로세스는 이 파일 시스템의 수퍼블럭에 대한 통제권을 내놓게 된다.

만약 파일 시스템에 프리 블럭이 충분하면 프로세스는 할당을 받게 된다. 만약 EXT2 파일 시스템이 데이터 블럭을 미리 할당하도록 만들어졌다면 미리 할당된 블럭을 사용할 수도 있 다. 미리 할당된 블럭은 실제로 존재하지는 않지만 할당된 블럭 비트맵에 예약되어 있다. 우 리가 새로운 데이터 블럭을

할당해 주려고 하는 파일을 나타내는 VFS inode는 EXT2 고유의 항목 두개를 갖고 있다. prealloc_block은 처음에 미리 할당된 데이터 블럭의 수를 나타 내고, prealloc_count는 그 중에서 몇 개가 남아 있는지를 나타낸다. 미리 할당된 블럭이 없거나 블럭을 미리 할당하는 기능이 사용되지 않고 있으면, EXT2 파일 시스템은 새로운 블 럭을 할당하여야만 한다. EXT2 파일 시스템은 우선 파일의 마지막 데이터 블럭의 다음 데이 터 블럭이 비었는지 본다. 논리적으로 보아 이 블럭은 순차식 액세스를 더욱 빠르게 해주기 때문에 가장 효율적인 블럭이다. 만약, 그 블럭이 비어있지 않다면 검색의 범위를 넓혀서 가 장 이상적인 블럭에서 64블럭 이내의 데이터 블럭을 살펴본다. 이러한 블럭은 비록 가장 이 상적이지는 않지만 충분히 가까우며 그 파일에 속한 다른 데이터 블럭과 같은 블럭 그룹에 속한다.

만약, 그러한 블럭 중에서도 빈 것이 없으면, 빈 블럭이 나타날 때 까지 다른 모든 블럭 그 룹을 뒤지게 된다. 블럭 할당 프로그램은 한 블럭 그룹 안에서 여덟개의 빈 데이터 블럭으 로 된 덩어리를 찾으려고 한다. 여덟개 짜리를 찾지 못하면 더 작은 것이라도 찾아야 한다. 만약 블럭 미리 할당 기능이 필요하고 사용가능하게 되어 있으면 prealloc_block 과 prealloc_count 값을 각기 갱신한다.

블럭 할당 프로그램은 빈 블럭을 찾을 때마다 블럭 그룹의 블럭 비트맵을 갱신하고 버퍼 캐 시 내에 데이터 버퍼를 할당한다. 그러한 데이터 버퍼는 파일 시스템을 지원하는 장치 식별 자와 할당된 블럭의 블럭 번호에 의하여 유일하게 식별된다. 버퍼내의 데이터가 모두 0이고 버퍼가 "더티(dirty)" 라고 표시되어 있으면 이는 실제 디스크에 내용이 기록되지 않았음을 나타낸다. 마지막으로 수퍼블럭의 내용이 바뀌었고 락이 되어 있지 않음을 나타내기 위하여 "더티(dirty)" 라고 표시한다. 수퍼블럭을 기다리는 있는 프로세스가 있었다면 큐에 들어 있 는 프로세스 중 첫번째 프로세스가 다시 실행되게 되며 파일 처리에 필요한 수퍼블럭의 독 점적 통제를 갖게 된다. 프로세스의 데이터는 데이터 블럭이 다 채워지면 또 새로운 데이터 블럭에 기록되며 이러한 과정은 데이터 블럭이 할당될 때마다 똑같이 반복된다.

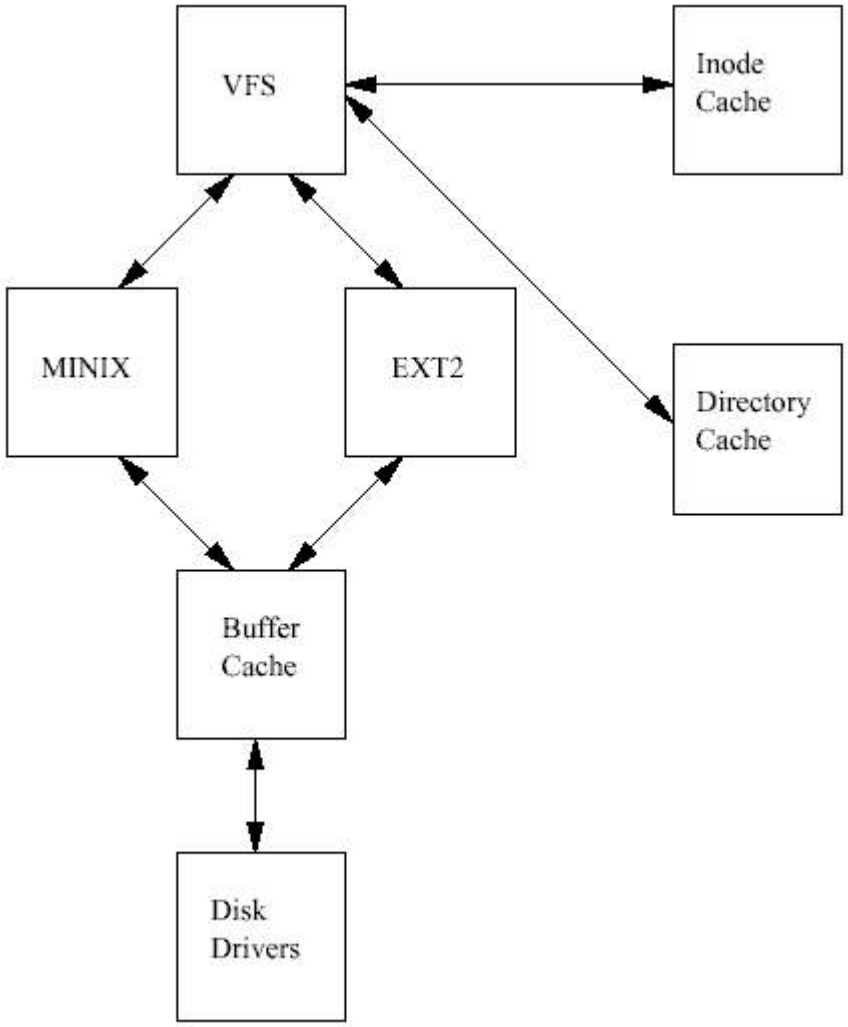

그림 9.4 : 가상 파일 시스템의 논리적 구성도

9.2 가상 파일 시스템(Virtual File System, VFS)

그림 9.4는 리눅스 커널의 가상 파일 시스템과 실제 파일 시스템과의 관계를 보여준다. 가상 파일 시스템은 어느 순간이든 마운트된 서로 다른 파일 시스템 모두를 다룰 수 있어야 한다. 이를 위해 전체 (가상) 파일 시스템과 실제 마운트된 파일 시스템을 기술하는 자료구조를 관리하여야 한다. 더 혼란스럽게 말하자면, VFS는 EXT2 파일 시스템이 수퍼블럭과 inode를 사용하는 것과 상당히 비슷한 방법으로, 시스템에 있는 파일을 수퍼블럭과 inode로 나타낸다. EXT2 inode처럼 VFS inode는 시스템에 있는 파일과 디렉토리 즉 가상 파일 시스템의 내용과 배치를 나타낸다. 이제부터 혼동을 피하기 위하여, EXT2의 inode와 수퍼블럭과는 달리 "VFS inode"와 "VFS 수퍼블럭"이라고 표기하도록 하겠다.

각 파일 시스템들은 초기화될때, 자신을 VFS에 등록한다. 이는 시스템 부팅중에 운영체제가 초기화되면서 일어난다. 실제 파일 시스템은 커널 자체에 포함되거나 모듈로 만들어진다. 파 일 시스템 모듈은 시스템이 필요로 할 때 로드된다. 예를 들어, VFAT 파일시스템이 커널 모 듈로 되어 있다면, VFAT 파일 시스템이 마운트될 때 로드될 것이다. 블럭장치에 기반한 파 일 시스템이 마운트되고, 이것이 루트 파일 시스템을 포함하고 있다면, VFS는 이것의 수퍼 블럭을 읽어야 한다. 파일 시스템 타입별 수퍼블

럭 읽기 루틴은 파일 시스템의 배치도를 정 확히 알 수 있어야 하며, 그 정보를 VFS 수퍼블럭 자료구조에 매핑 시킬 수 있어야 한다. VFS는 마운트된 파일 시스템과 VFS 수퍼블럭의 리스트를 관리한다. 각각의 VFS 수퍼블럭 은 특정 기능을 수행하는 루틴에 대한 정보와 포인터를 갖고 있다. 따라서, 예를 들어 마운 트된 EXT2 파일 시스템을 나타내는 수퍼블럭은 EXT2 고유의 inode 읽기 루틴에 대한 포인터를 갖고 있다. 이 EXT2 inode 읽기 루틴은 다른 모든 파일 시스템 고유의 inode 읽기 루틴 과 마찬가지로 VFS inode의 각 항목을 채운다. 각각의 VFS 수퍼블럭은 파일 시스템의 첫번 째 VFS inode에 대한 포인터를 갖고 있다. 루트 파일 시스템의 경우 이 inode는 "/" 디렉토리 를 나타낸다. 이러한 정보의 매핑은 EXT2 파일 시스템의 경우에는 매우 효율적이지만 다른 파일 시스템에서는 좀 덜 효율적이다.

시스템의 프로세스가 디렉토리나 파일을 액세스하려고 하면 시스템내의 VFS inode를 탐색하 는 시스템 루틴을 부르게 된다. 예를 들어, 어떤 디렉토리에 대해 ls 명령을 치거나 어떤 파 일에 대하여 cat 명령을 치면, 가상 파일 시스템은 파일 시스템을 나타내는 VFS inode들을 주욱 찾아나가게 된다. 시스템에 있는 모든 파일이나 디렉토리는 각기 하나의 VFS inode에 의하여 표현되므로 수많은 inode가 반복적으로 액세스되게 된다. 액세스 속도를 빠르게 하기 위하여 이들 inode는 inode 캐시에 저장된다. 어떤 inode가 inode 캐시에 들어있지 않으면 해 당 inode를 읽어들이기 위하여 각 파일 시스템 고유의 루틴을 호출하여야 한다. 이렇게 읽어 들인 inode는 inode 캐시에 저장되어 다음번 액세스할 때에는 캐시에서 찾을 수 있게 된다. 덜 사용되는 VFS inode는 캐시로부터 제거된다.

모든 리눅스 파일 시스템은 파일 시스템을 갖고 있는 실제 장치에 대한 액세스 속도를 높이 기 위하여 공통의 버퍼 캐시를 사용한다. 이 버퍼 캐시는 파일 시스템과는 상호 독립적이며 리눅스 커널이 데이터 버퍼를 할당하고 읽고 쓰는 메커니즘에 통합되어 있다. 리눅스 파일 시스템을 그 아래에 있는 매체나 그를 지원하는 장치로부터 독립적으로 만드는 것은 뚜렷한 장점을 가져다 준다. 모든 블럭 구조의 장치는 리눅스 커널에 자신을 등록하며 통일되고, 블 럭 기반의, 일반적으로는 비동기적인 인터페이스를 제공한다. 심지어 SCSI 장치와 같이 비 교적 복잡한 블럭장치도 이렇게 한다. 실제 파일 시스템이 그 아래에 깔려있는 실제 디스크 에서 데이터를 읽게 되면 이는 블럭 디바이스 드라이버로 하여금 자신이 컨트롤하는 장치로 부터 실제 블럭을 읽도록 요청하는 것이 된다. 이러한 블럭 장치 인터페이스에 버퍼 캐시는 통합되어 있다. 파일 시스템이 어떤 블럭을 읽으면 그 블럭은 전체 버퍼 캐시에 저장되어 모든 파일 시스템과 리눅스 커널에 의하여 공유된다. 그 안에 있는 버퍼 각각은 블럭 번호와 그 블럭을 읽은 장치의 고유 식별자에 의하여 구분된다. 따라서, 같은 데이터가 자주 필 요하게 되면 시간이 많이 걸리는 실제 디스크에서 읽는 것이 아니라 버퍼 캐시에서 꺼내서 쓰게 된다. 어떤 장치는 혹시 필요할 경우를 대비하여 데이터 블럭을 미리 읽어두는 미리 읽기(read ahead) 기능을 지원한다.

VFS에서는 자주 사용되는 디렉토리의 inode를 빨리 찾기 위하여 디렉토리 찾아보기 캐시도 갖고 있다. 실험삼아 최근에 리스트를 본 적이 없는 디렉토리의 리스트를 보려고 해보라. 처 음에 볼 때에는 약간 멈칫 한 후에 리스트가 나오지만 두번째부터는 곧바로 나오게 된다. 디렉토리 캐시에는 디렉토리 그 자체에 대한 inode를 저장하는 것이 아니다. 이러한 inode는 inode 캐시에 저장된다. 디렉토리 캐시는 단지 전체 디렉토리 이름과 그에 해당하는 inode 번호와의 매핑을 저장한다.

9.2.1 VFS 수퍼블럭

마운트된 파일 시스템은 VFS 수퍼블럭에 의해 표현된다. 다른 여러가지 정보도 있지만 VFS 수퍼블럭에서 눈여겨 볼 만한 정보는 다음과 같다.

- **장치(Device)** 이것은 이 파일 시스템이 저장되어 있는 블럭 장치의 장치 식별자이다. 예를 들어 시스템의 첫번째 IDE 하드 디스크인 /dev/hda1은 장치 식별자로 0x301을 갖는다.
- **inode 포인터** mounted inode 포인터는 이 파일 시스템의 첫번째 inode를 가리킨다. covered inode 포인터는 이 파일 시스템이 마운트된 디렉토리를 표현하는 inode를 가리킨다. 루트 파일 시스템의 VFS 수퍼블럭은 covered inode 포인터가 없다.
- **블럭 크기(Blocksize)** 블럭 크기는 이 파일 시스템의 블럭의 크기를 바이트 단위로 - 예를 들 어, 1024 바이트[3] - 나타낸 것이다.
- **수퍼블럭 연산(Superblock Operations)** 이 파일 시스템에 대한 수퍼블럭 루틴의 집합이다. 다른 용도로 사용되기도 하지만, 이들 루틴은 VFS가 inode와 수퍼블럭을 읽고 쓰기 위해 사용된다.
- **파일 시스템 타입(File System Type)** 마운트된 파일 시스템의 file_system_type 자료구조 를 가리키는 포인터이다.
- **파일 시스템 고유 정보(File System Specific)** 이 파일 시스템이 필요로 하는 정보를 가리키는 포인터.

9.2.2 VFS inode

EXT2 파일 시스템과 마찬가지로, VFS 안에 있는 모든 파일, 디렉토리 등은 반드시 단지 하 나의 VFS inode로 표현된다[4]. 각 VFS inode의 정보는 파일 시스템의 정보로부터 파일 시스템 고유 루틴에 의해 생성된다. VFS inode는 커널의 메모리에만 존재하고, 시스템에서 필요한 동안에만 VFS inode 캐시에 저장되어 있다. 다른 여러가지 정보도 있지만 VFS inode에서 눈 여겨 볼 만한 정보는 다음과 같다.

- **장치(Device)** 이것은 이 VFS inode가 나타내는 파일 또는 다른 어떤 것을 가지고 있는 장치 의 장치 식별자이다.
- **inode 번호** 이것은 inode의 번호이고, 이 파일 시스템 안에서 유일하다. 장치와 inode 번호의 조합은 VFS 내에서 유일하다.
- **모드(Mode)** EXT2와 마찬가지로 이 항목은 이 VFS inode가 무엇(파일, 디렉토리, 기타)을 나 타내는가와 접근 권한 등을 나타낸다[5].
- **사용자 식별자(user id)** 소유자를 나타낸다.
- **시각(times)** 생성시간, 변경시간, 읽은 시간 등을 나타낸다.
- **블럭 크기(block size)** 이 파일의 블럭 크기를 바이트 단위 - 예를 들어, 1024 바이트 - 로 나 타낸다.
- **inode 연산(inode operations)** 연산 루틴의 주소들에 대한 포인터이다. 이들 루틴은 파일 시스템마다 고유하며 inode에 대한 여러가지 연산, 예를 들어 이 inode가 나타내는 파일의 제 거과 같은 일을 수행한다.

- **사용횟수(count)** 이 VFS inode를 현재 사용하는 시스템 요소의 수이다. count가 0이면 inode가 프리이며 제거되거나 재사용될 수 있다.
- **락(lock)** VFS inode를 락을 걸기 위해 사용한다. 예를 들어 파일 시스템에서 이 inode를 읽을 때 사용된다.
- **더티(dirty)** 이 VFS inode가 기록된 적이 있는가 즉, 하부 파일 시스템도 변경이 필요한 가를 나타낸다.
- **파일 시스템 고유 정보.(file system specific information)**

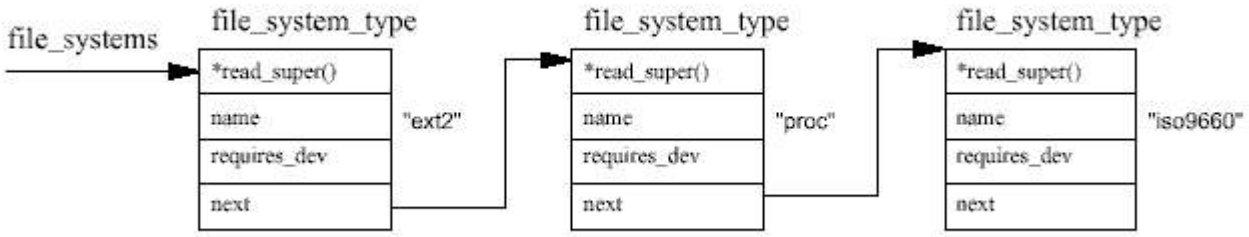

그림 9.5 : 등록된 파일 시스템

9.2.3 파일 시스템 등록하기

리눅스 커널을 빌드할 때 어떤 파일 시스템을 지원할 것인지 지정할 수 있다. 커널을 빌드 할 때, 파일 시스템 시작 코드는 내장된 모든 파일 시스템의 초기화 루틴을 호출한다. 리눅 스 파일 시스템은 모듈로 만들어질 수도 있는데, 이 경우에는 필요할 때 로드되거나, insmod에 의해 수작업으로 로드된다. 파일 시스템 모듈은 로드될 때마다 자신을 커널에 등 록하고, 언로드될 때 자신을 해제한다. 각 파일 시스템의 초기화 루틴은 자신을 가상 파일 시스템(VFS)에 등록하며, file_system_type 자료구조에 의해 표현된다. 자료구조에는 파 일 시스템의 이름과 VFS 수퍼블럭 읽기 루틴에 대한 포인터가 저장되어 있다. 그림 9.5는 file_system_type 자료구조가 file_systems 포인터가 가리키는 리스트로 저장되어 있 는 것을 보여준다. 각 file_system_type 자료구조는 다음 정보를 포함한다.

- **수퍼블럭 읽기 루틴** 이 루틴은 파일 시스템이 마운트될 때 VFS에 의해 호출된다.
- **파일 시스템 이름** 이 파일 시스템의 이름으로 예를 들어 ext2.
- **필요한 장치** 이 파일 시스템을 실제로 지원하는 장치가 필요한가를 나타낸다. 모든 파일 시 스템이 저장될 장치가 필요한 것은 아니다. 예를 들어, /proc 파일 시스템은 블럭 장치 가 필요로 하지 않는다.

어떤 파일 시스템이 등록되어 있는지는 /proc/filesystems의 내용을 보면 알 수 있다. 예를 들면 다음과 같다.

```
        ext2
nodev   proc
        iso9660
```

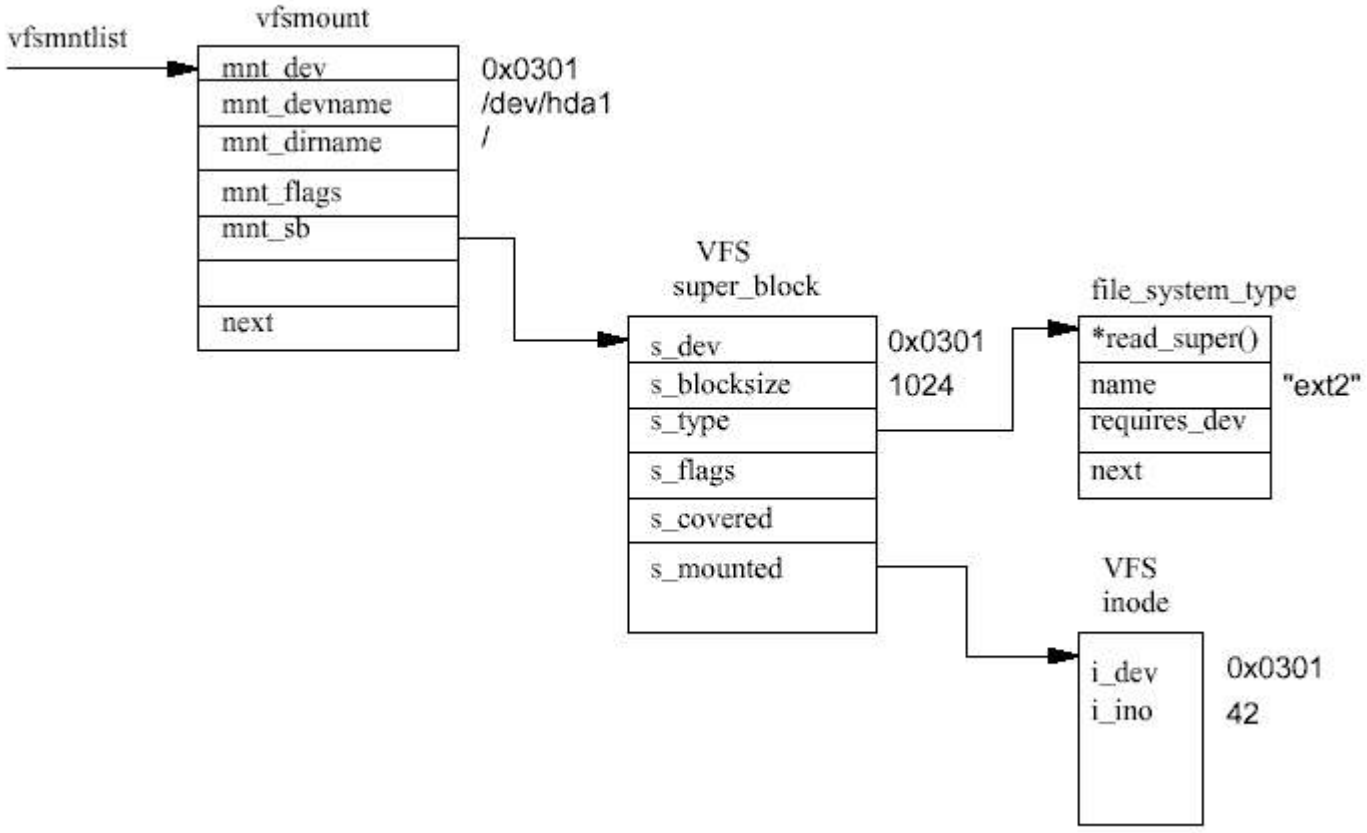

그림 9.6 : 마운트된 파일 시스템

9.2.4 파일 시스템 마운트하기

수퍼유저가 파일 시스템을 마운트하려고 할 때, 리눅스 커널은 시스템 콜로 전달된 인자가 옳은지 확인해야 한다. mount가 기본적인 검사를 하긴 하지만, 커널이 어떤 파일 시스템을 지원하도록 빌드되었는지, 마운트 지점이 실제로 존재하는지는 알지 못한다. 다음 mount 명령을 예로 살펴보자.

mount -t iso9660 -o ro /dev/cdrom /mnt/cdrom

이 mount 명령은 커널에 세 가지 정보를 전달한다. 파일 시스템의 이름, 파일 시스템을 포 함하고 있는 블럭 장치, 그리고 새로운 파일 시스템이 현재의 파일 시스템 배치도의 어디에 마운트될 것인가 하는 것이다.

가상 파일 시스템이 반드시 해야 하는 첫번째 일은 파일 시스템을 찾는 것이다. 이를 위해 알려진 파일 시스템들의 리스트를 탐색한다. 즉 file_systems가 가리키는 리스트에서 각 각의 file_system_type 자료구조를 살펴본다. 일치하는 이름을 찾는다면, 이 파일 시스 템 타입은 커널이 지원하는 것이고, 커널이 이 파일 시스템의 수퍼블럭을 읽는 파일 시스템 고유의 루틴의 주소를 갖고 있다는 것이다. 일치하는 파일 시스템 이름을 찾지 못하더라도, 커널이 커널 모듈을 요구시 로드하도록 빌드되었다면(12장을 참조) 아직 완전히 실패한 것 은 아니다. 이 경우에는 커널은 작업을 계속하기 전에 커널 데몬이 적절한 파일 시스템 모 듈을 로드하도록 요청한다.

다음으로 만약 mount에 전달된 물리적 장치가 아직 마운트되지 않았다면, 새로운 파일 시 스템의 마운트 지점이 될 디렉토리의 VFS inode를 찾아야 한다. 이 VFS inode는 inode 캐시 에 있거나, 아니면 마운트 지점의 파일 시스템을 저장하고 있는 블럭 장치에서 읽어야 한다. inode를 찾으면 이것이 디렉토리인지, 그리고 여기에 이미 다른 파일 시스템이 마운트된 것 은 아닌지 검사한다. 한 디렉토리는 단 하나의 파일 시스템의 마운트 지점으로만 사용될 수 있다.

이 시점에서 VFS 마운트 코드는 새로운 VFS 수퍼블럭을 할당하고 이를 마운트 정보와 함 께 이 파일 시스템을 위한 수퍼블럭 읽기 루틴에 전달한다. 시스템의 모든 VFS 수퍼블럭은 super_block 자료구조의 super_blocks 벡터에 저장된다. 그리고 이번 마운트를 위해 그 중의 하나가 할당된다. 수퍼블럭 읽기 루틴은 물리적 장치에서 읽은 정보에 따라 VFS 수퍼블럭을 채워야 한다. EXT2 파일 시스템의 경우 이 정보의 매핑 또는 변환은 매우 쉽다. 단지 EXT2 수퍼블럭을 읽고 VFS 수퍼블럭을 채우면 된다. 다른 파일 시스템들, 예를 들어 MS DOS 파일 시스템의 경우 이것은 그리 쉬운 일은 아니다. 어떤 파일 시스템이든, VFS 수 퍼블럭을 기록한다는 것은 그 파일 시스템으로 된 블럭 장치에서 무엇이든 읽을 수 있다는 것을 의미한다. 만약 블럭 장치로부터 읽지 못한다면, 즉 블럭 장치가 이 타입의 파일 시스템으로 되어 있지 않으면 mount 명령은 실패하게 된다.

마운트된 각 파일 시스템은 vfsmount 자료구조로 기술된다 (그림 9.6 참조). 이들은 vfsmntlist가 가리키는 리스트에 큐되어 있다. vfsmnttail 포인터는 리스트의 마지막 항 목을 가리키고, mru_vfsmnt 포인터는 가장 최근에 사용된 파일 시스템을 가리킨다. 각 vfsmount 구조는 파일 시스템을 담고있는 블럭 장치의 장치 번호, 이 파일 시스템이 마운 트된 디렉토리, 이 파일 시스템이 마운트될 때 할당된 VFS 수퍼블럭에 대한 포인터 등을 갖고 있다. VFS 수퍼블럭은 해당 종류의 파일 시스템에 대한 file_system_type 자료구조 와 이 파일 시스템의 루트 inode를 가리킨다. 이 inode는 이 파일 시스템이 로드되어 있는 동안 VFS inode 캐시에 항상 존재한다.

9.2.5 가상 파일 시스템(VFS)에서 파일 찾기

가상 파일 시스템에서 어떤 파일의 VFS inode를 찾으려면, VFS는 파일 이름을 구성하는 중 간 디렉토리를 나타내는 VFS inode를 한번에 하나씩 찾아가며 디렉토리 이름을 해석해야 한 다. 각 디렉토리를 검색하는 과정에서 파일 시스템 고유의 검색 함수를 호출하게 되며, 이 함수의 주소는 부모 디렉토리를 나타내는 VFS inode에 저장되어 있다. 이것이 가능한 이유 는 항상 각 파일 시스템의 루트의 VFS inode를 알고 있고, 이것은 파일 시스템의 VFS 수퍼 블럭에서 가리키고 있기 때문이다. 실제 파일 시스템이 어떤 inode를 검색할 때, 각 디렉토 리에 대해 디렉토리 캐시를 검사한다. 디렉토리 캐시에 항목이 없으면, 실제 파일 시스템은 기반하는 파일 시스템이나 inode 캐시에서 VFS inode를 가져온다.

9.2.6 가상 파일 시스템에서 파일 만들기

9.2.7 파일 시스템의 마운트 해제

보통 조립은 분해의 역순이라고 한다. 이 말은 파일 시스템의 마운트 해제(unmount)에도 어 느정도 적용된다. 파일 시스템의 마운트를 해제하려면 시스템에서 그 파일 시스템내의 파일 을 사용하고 있는 것이 없어야 한다. 따라서 어떤 프로세스가 /mnt/cdrom 디렉토리나 그 아래 디렉토리를 사용하고

있다면 마운트를 해제할 수 없다. 만약 무엇인가가 마운트를 해 제하려는 파일 시스템을 사용하고 있다면, VFS inode 캐시에 그 파일 시스템에 속하는 VFS inode가 들어 있을 것이다. 따라서 마운트 해제 프로그램은 해제하려는 파일 시스템이 차지 하고 있는 장치에 속하는 inode가 캐시의 inode 리스트에 들어 있는지 검사한다. 마운트된 파일 시스템의 VFS 수퍼블럭이 더티하면, 즉 내용이 수정되었다면, 수퍼블럭을 디스크의 파 일 시스템에 기록하여야만 한다. 일단 디스크에 기록하고 나면 VFS 수퍼블럭이 차지하고 있던 메모리를 커널의 메모리 풀에 보내준다. 그리고 마지막으로 이 파일 시스템의 마운트 에 필요했던 vfsmount라는 데이터 구조를 vfsmntlist로 부터 떼어낸 다음 해제한다.

9.2.8 VFS inode 캐시

마운트된 파일 시스템을 뒤질 때 그에 해당하는 VFS inode를 계속 읽거나 쓰게 된다. 가상 파일 시스템은 마운트된 파일 시스템에 대한 액세스 속도를 높이기 위하여 inode 캐시를 유 지한다. VFS inode를 inode 캐시에서 읽을 수 있다면 그만큼 실제 장치에 대한 액세스를 덜 해도 된다.

VFS inode 캐시는 해시 테이블로 구현되었으며, 테이블 내의 각 엔트리는 같은 해시 값을 갖는 VFS inode의 리스트를 가리키고 있다. inode의 해시 값은 inode 번호와 그 파일 시스템 을 갖고 있는 실제 장치의 장치 식별자로부터 계산된다. 가상 파일 시스템이 inode를 액세스 할 필요가 있을 때 마다 VFS inode 캐시를 먼저 찾아본다. 캐시내의 inode를 찾기 위해서 시 스템은 먼저 해시 값을 계산하고 그 값을 인덱스로하여 inode 해시 테이블을 본다. 그러면 같은 해시 값을 가진 inode 리스트에 대한 포인터를 얻게 된다. 이 리스트에서 찾으려는 것 과 같은 inode 번호와 장치 식별자를 가진 inode가 나타날 때까지 각각의 inode를 살펴본다.

만약 캐시에서 inode를 찾게되면 카운트 값을 증가시킴으로써 그 inode를 사용하는 사용자가 있다는 것을 알려준 다음 파일 시스템에 대한 액세스를 계속한다. 만약 찾을 수 없다면 파 일 시스템이 메모리로부터 inode를 읽을 수 있도록 빈 VFS inode를 찾아야만 한다. VFS가 빈 inode를 찾는데에는 여러 가지 방법이 있다. 만약 시스템이 VFS inode를 더 할당할 수 있다면 다음과 같은 방법을 쓰게 된다 - 커널 페이지를 할당하고 이를 여러 개의 새로운 빈 inode 로 쪼갠다음 inode 리스트에 넣는다. 시스템에 있는 모든 VFS inode는 first_inode가 가리 키는 리스트와 inode 해시 테이블에 들어있게 된다. 만약 시스템에 허용된 만큼 모든 inode 를 이미 할당하였다면 재사용할만한 inode 후보들을 찾아야만 한다. 사용 횟수가 0인 inode 는 현재 시스템에서 사용되고 있지 않다는 의미이므로 좋은 후보가 된다. 정말로 중요한 VFS inode, 예를 들어 파일 시스템의 루트 inode는 사용 횟수가 0보다 훨씬 큰 값이므로 재 사용의 후보가 되는 경우가 결코 없다. 일단 재사용 후보가 선택되면 그 내용을 깨끗이 지 운다. 만약 VFS inode가 더티하면 파일 시스템에 그 내용을 기록할 필요가 있으며, 만약 락 이 되어 있다면 락이 풀릴 때까지 기다려야 한다. 후보 VFS inode는 재사용되기 전에 반드 시 깨끗이 하여야 한다.

어쨌든 새로운 VFS inode를 발견하면 파일 시스템은 실제 파일 시스템에서 읽어온 정보를 inode에 채우는 특정 루틴을 부른다. inode를 채우는 동안 그 새 VFS inode의 사용 횟수는 1 이 되고 락이 되기 때문에, 그 inode가 완전한 정보를 갖게 될 때까지는 아무도 액세스 할 수 없다.

실제로 필요한 VFS inode를 얻기 위해서 그 외 다른 여러개의 inode를 액세스할 필요가 있 다. 디렉토리를 읽을 때에 이러한 일이 발생한다. 최종 디렉토리의 inode가 우리가 실제로 필요로 하는 것이지만, 그것을 얻기 위해서는 그 중간 디렉토리들의 inode도 읽어야만 한다. VFS inode 캐시가 사용되어 꽉 차게 되면, 덜 사용되는 inode는 버려지고 더 많이 사용되는 inode는 캐시에 남게 된다.

9.2.9 디렉토리 캐시(Directory Cache)

흔히 쓰이는 디렉토리에 대한 액세스 속도를 높이기 위해, VFS에서는 디렉토리 엔트리에 대 한 캐시를 유지한다. 디렉토리는 실제 파일 시스템에 의하여 참조되므로 실제 파일 시스템 에 대한 내용도 디렉토리 캐시에 저장된다. 다음 번에 똑같은 디렉토리가 참조되면 (예를 들어, 어떤 디렉토리의 리스트를 본 다음 그 리스트에 있는 어떤 파일을 연다면) 디렉토리 캐시에서 정보를 꺼낼 수 있다. 짧은 이름(최대 15자까지)을 가진 디렉토리 엔트리만 캐시가 되는데 이는 짧은 디렉토리 이름이 더 자주 사용되기 때문이다. 예를 들어, X 서버가 실행중 이라면 /usr/X11R6/bin 디렉토리는 매우 자주 액세스될 것이다.

디렉토리 캐시는 해시 테이블로 구성되는데, 이 테이블에서 각각의 엔트리는 같은 해시 값 을 가진 디렉토리 캐시 엔트리들의 리스트를 가리키고 있다. 해시 함수는 파일 시스템을 갖 고 있는 장치의 장치 번호와 디렉토리 이름을 이용하여 해시 테이블 내의 위치 즉 인덱스를 산출해낸다. 이렇게 함으로써 캐시된 디렉토리 엔트리를 빨리 찾을 수 있다. 엔트리를 찾는 데 시간이 너무 많이 걸리거나 심지어 찾을 수 없다면 캐시를 사용할 필요가 없을 것이다.

캐시 값을 유효하게 하고 최신의 값으로 유지하기 위하여 VFS는 LRU(최근에 가장 적게 사 용된, Least Recently Used) 방식으로 디렉토리 캐시 엔트리 리스트를 관리한다. 디렉토리 엔트 리는 처음으로 참조되어 캐시로 들어갈 때 1단계 LRU 리스트의 맨 뒤로 가서 붙게 된다. 만약 캐시가 가득 차 있으면 LRU 리스트의 맨 앞 엔트리를 대치한다. 디렉토리 엔트리가 다시 한번 액세스되면 2단계 LRU 캐시 리스트로 올라가게 된다. 물론 이런 경우에는 2단계 LRU 캐시 리스트의 앞쪽에서 디렉토리 엔트리를 대치하며 들어갈 수도 있다. 1단계와 2단계 의 LRU 리스트에서 맨 앞의 엔트리를 대치하는 것은 제대로 된 것이다. 어떤 엔트리가 리 스트의 맨 앞에 나와 있다는 것은 최근에 액세스 된 적이 없다는 것을 의미하기 때문이다. 만약에 최근에 액세스된 적이 있다면 리스트의 뒤쪽 어딘가에 있어야 할 것이다. 2단계 LRU 캐시 리스트에 들어 있는 엔트리들은 1단계 LRU 캐시 리스트에 들어 있는 엔트리들보다 안 전하다. 엔트리가 2단계 리스트에 들어있다는 것은 액세스되었을 뿐만 아니라 반복적으로 참조되고 있음을 의미하기 때문에 더 안전하게 보관할 필요가 있다.

REVIEW NOTE : 그림이 필요한가?

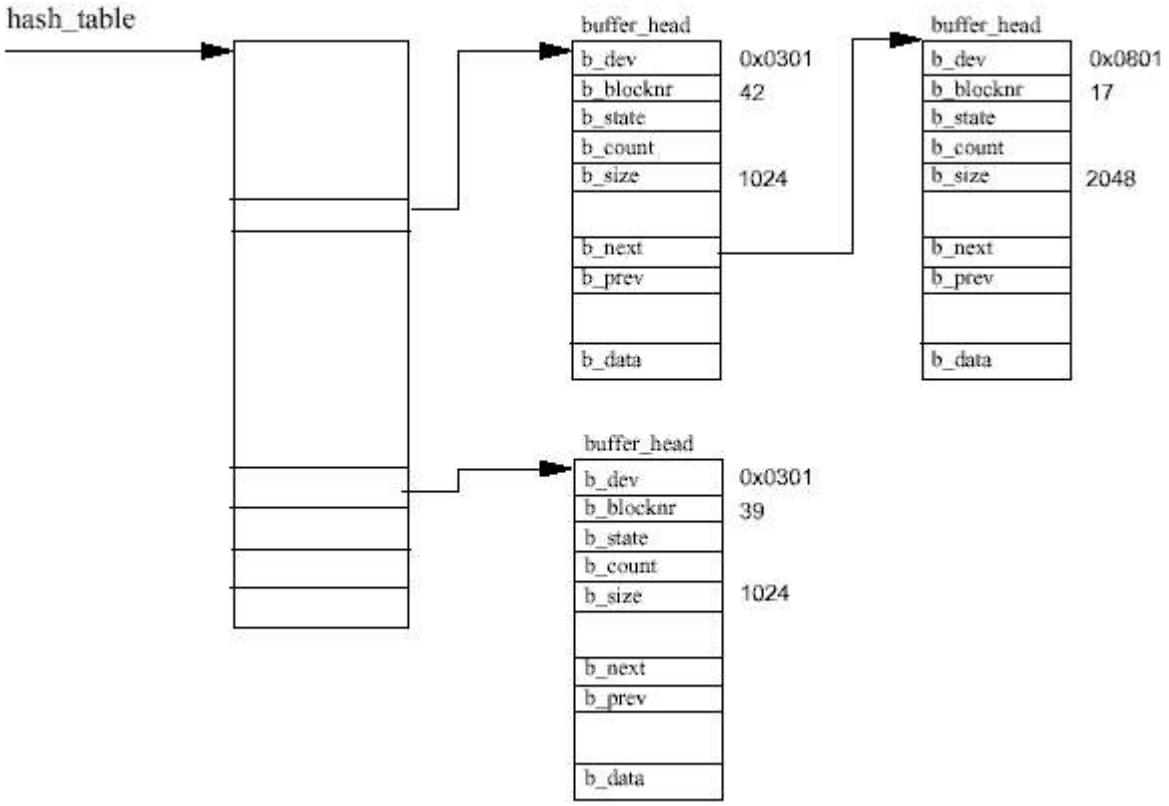

그림 9.7 : 버퍼 캐시

9.3 버퍼 캐시(Buffer Cache)

마운트된 파일 시스템을 사용하게 되면 이는 블럭 장치에서 데이터 블럭을 읽거나 쓰는 많 은 요구가 발생하게 된다. 데이터 블럭을 읽고 쓰는 모든 요구들은 표준 커널 함수 호출을 통하여 디바이스 드라이버에 buffer_head 자료구조의 형태로 전달된다. 이 자료구조는 블 럭 디바이스 드라이버가 필요로 하는 모든 정보를 제공한다. 장치 식별자는 장치를 유일하 게 구별해주고, 블럭 번호는 드라이버가 어떤 블럭을 읽어야 하는지 말해준다. 모든 블럭장 치는 똑같은 크기의 블럭들이 선형으로 모여진 것처럼 보인다. 물리적인 블럭 장치로의 접 근 속도를 빠르게 하기 위해 리눅스는 블럭 버퍼 캐시를 관리한다. 시스템에 있는 모든 블 럭 버퍼들은 새 것이던, 안쓰이는 버퍼이던간에 이 버퍼 캐시 어디엔가 존재한다. 모든 물리 적인 블럭 장치들은 이 캐시를 공유하며, 어떤 순간이던지 캐시에는 많은 블럭 버퍼가 시스 템에 있는 블럭 장치 중의 하나에 소속되어 각자 서로 다른 상태에 있을 것이다. 버퍼 캐시 에 올바른 데이터가 있다면, 이는 시스템이 물리적인 장치에 접근하는 것을 절약해준다. 블 럭 장치로부터 데이터를 읽는데 사용하거나 쓰는데 사용한 어떤 블럭 버퍼이든지간에 버퍼 캐시로 들어간다. 시간이 지나면 이들은 마땅히 캐시에 있을만한 버퍼를 위해 자리를 내주 던지, 자주 사용된다면 캐시에 계속 남아 있게 된다.

캐시에 있는 블럭 버퍼는 이를 소유하는 장치 식별자와 버퍼의 블럭번호로 유일하게 구별된 다. 버퍼 캐시는 두개의 기능적인 부분으로 되어있다. 첫번째 부분은 프리 블럭 버퍼의 리스 트이다. 지원하는 버퍼 크기별로 각기 하나의 리스트가 있고, 시스템의 프리 블럭 버퍼는 처 음 만들어질 때나 버려질 때 이들 리스트에 들어가게 된다. 현재 지원하는 버퍼의 크기는 512, 1024, 2048, 4096, 그리고 8192 바이트이다. 두번째 기능적인 부분은 캐시 그 자체이다. 이것은 해시 테이블로서 똑같은 해시 인덱스

를 가지는 버퍼들을 고리로 가리키고 있는 포인 터들의 벡터이다. 해시 인덱스는 해당 장치 식별자와 데이터 블럭의 블럭 번호로부터 만들 어진다. 그림 9.7은 몇개의 엔트리를 해시 테이블과 함께 보여주고 있다. 블럭 버퍼는 프리 리스트 중의 어떤 하나의 리스트 또는 버퍼 캐시 둘 중의 하나에 들어 있다. 이들이 버퍼 캐시에 있을 때 이들은 LRU 리스트에도 들어가게 된다. 각 버퍼 유형마다 LRU 리스트가 있고, 이들은 시스템이 특정 유형의 버퍼에 대해 일 - 예를 들어 새로운 데이터를 가진 버 퍼를 디스크에 기록하기 - 을 수행하는데 사용된다. 버퍼의 유형은 버퍼의 상태를 반영하며, 리눅스는 현재 다음과 같은 유형을 지원한다 :

- **깨끗한(clean)** 사용하지 않은, 새 버퍼
- **락되어있는(locked)** 버퍼에 락이 걸려 있으며, 기록되기를 기다리고 있다.
- **더티한(dirty)** 더티 버퍼. 이들은 새롭고 유효한 데이터를 가지고 있으며, 기록될 것이지만, 아직까지 언제 기록될 지 스케쥴이 잡히지 않았다.
- **공유(shared)** 공유 버퍼
- **공유하지않는(unshared)** 예전엔 공유했으나 이제는 공유하지 않는 버퍼

파일 시스템이 아래 계층의 물리적인 장치로부터 버퍼를 읽을 필요가 있을 때마다 버퍼 캐 시로부터 블럭을 얻으려고 시도한다. 만약 버퍼 캐시에서 버퍼를 얻을 수 없다면, 프리 리스 트에서 적당한 크기의 깨끗한 버퍼를 하나 얻게 되며, 이 새 버퍼는 버퍼 캐시에 들어가게 된다. 필요로 하는 버퍼가 버퍼 캐시에 있다면, 이것은 최근 것일수도 그렇지 않을 수도 있 다. 만약 최근 것이 아니거나, 새 블럭 버퍼라면, 파일 시스템은 디바이스 드라이버에게 해 당하는 데이터 블럭을 디스크에게 읽어오도록 한다.

다른 캐시와 마찬가지로, 버퍼 캐시는 효율적으로 동작하도록 관리되어야 하며, 버퍼 캐시를 사용하는 블럭 장치들 사이에서 공평하게 캐시 엔트리를 할당해야 한다. 리눅스는 bdflush 커널 데몬을 사용하여, 캐시에 대한 잡다한 일들을 수행하지만, 어떤 것들은 캐시를 사용한 결과로 자동적으로 일어난다.

9.3.1 bdflush 커널 데몬

bdflush 커널 데몬은 시스템이 너무 많은 더티 버퍼 - 언젠가는 디스크에 쓰여져야 하는 데이터를 가지고 있는 버퍼 - 를 가지게 되었을 때 동적으로 반응하는 간단한 커널 데몬이 다. 이는 시스템이 시작할 때 커널 쓰레드로서 시작되며, 혼동되지 않도록 자신을 kflushd라 고 부른다. 이 이름은 시스템에 있는 프로세스들을 살펴보기 위해 ps 명령을 썼을 때 볼 수 있는 이름이다. 대부분 이 데몬은 시스템에 있는 더티 버퍼의 갯수가 충분히 많아지기를 기 다리며 잠들어있다. 버퍼가 할당되거나 버려질 때 시스템에 있는 더티 버퍼의 갯수를 검사 한다. 만약 시스템에 있는 전체 버퍼의 갯수 중에서 더티 버퍼의 비율이 너무 커지면 bdflush가 깨어난다. 기본값으로 설정된 값은 60%이지만, 시스템에서 버퍼가 필요하다면 bdflush는 언제든지 깨어날 수 있다. 이 값은 update 명령으로 보거나 바꿀 수 있다 :

```
# update -d

bdflush version 1.4
0:      60      Max fraction of LRU list to examine for dirty blocks
1:      500     Max number of dirty blocks to write each time bdflush activated
2:      64      Num of clean buffers to be loaded onto free list by refill_freelist
```

```
3:      256     Dirty block threshold for activating bdflush in refill_freelist
4:      15      Percentage of cache to scan for free clusters
5:      3000    Time for data buffers to age before flushing
6:      500     Time for non-data (dir, bitmap, etc) buffers to age before flushing
7:      1884    Time buffer cache load average constant
8:      2       LAV ratio (used to determine threshold for buffer fratricide).
```

데이터를 버퍼에 써서 버퍼가 더티하게 되면 그 버퍼는 BUF_DIRTY LRU 리스트에 연결되 고, bdflush 는 이중에서 적당한 개수를 해당 디스크에 쓰려고 한다. 이 숫자 역시 update 명령으로 보고 제어할 수 있으며, 기본값은 500이다 (위에서 보는 바처럼).

9.3.2. update 프로세스

update 명령은 단순히 명령만이 아니라, 데몬이기도 하다. 수퍼유저로서 실행되면 (시스템 초기화동 안에), 주기적으로 오래된 더티 버퍼들을 모두 디스크에 기록한다. 이는 bdflush 하고 유사한 일을 하는 시스템 서비스 루틴을 부름으로써 이루어지게 된다. 더티 버퍼가 다 쓰여지고 나면 그 때의 시스템 시간을 표시해 둔다. update는 실행될 때마다 시스템에 있는 모든 더티 버퍼에서 시간이 만료된 것들을 찾는다. 만료된 모든 버퍼는 디스크에 기록된다.

9.4 /proc 파일 시스템

/proc 파일 시스템이라말로 리눅스 가상 파일 시스템의 힘을 보여주는 것이다. 이는 실제 로 존재하는 것이 아니다 (리눅스의 또다른 마술같은 기교이다). /proc 디렉토리도, 이의 서 브 디렉토리도, 파일들로 실제로 존재하지 않는다. 그렇다면 어떻게 cat /proc/devices를 할 수 있는가? /proc 파일 시스템은 실제 파일 시스템과 마찬가지로 자신을 가상 파일 시 스템에 등록한다. 그러다가 파일이나 디렉토리를 열면서 VFS가 inode를 요청하면, /proc 파 일 시스템은 이들 파일과 디렉토리를 커널에 있는 정보를 가지고 만들어낸다. 예를 들어, 커 널의 /proc/devices 파일은 장치들을 나타내는 커널 자료구조로부터 생성된다.

/proc 파일 시스템은 사용자에게 커널의 내부 작업을 볼 수 있는 있는 창을 제공한다. 12장 에서 설명하고 있는 리눅스 커널 모듈같은 어떤 리눅스 서브시스템들은 /proc 파일 시스템 에 엔트리를 생성하기도 한다[6].

9.5 장치 특수 파일(Device Special Files)

리눅스는 다른 모든 버전의 유닉스와 마찬가지로 하드웨어 장치들을 특수 파일로 보여준다. 예를 들어 /dev/null은 널(null) 장치이다. 장치 파일은 파일 시스템에서 아무런 데이터 영 역도 차지하지 않는다. 이는 단지 디바이스 드라이버로의 접근점일 뿐이다. EXT2 파일 시스 템과 리눅스 VFS는 모두 장치 파일을 inode의 특수한 유형으로 구현한다. 장치 파일에는 문 자 특수 파일과 블럭 특수 파일이라는 두가지 형태가 있다. 커널 안에서, 디바이스 드라이버 는 파일처럼 구현되어 있다. 즉, 이를 열고, 닫는 등의 일을 할 수 있다. 문자 장치는 문자모 드로 I/O 작업을 할 수 있으며, 블럭 장치는 모든 I/O가 버퍼 캐시를 통하도록 되어 있다. 장치 파일로 I/O 요구를 하면, 이는 시스템 내에 있는 해당하는 디바이

스 드라이버로 전달 된다. 종종 이는 실제 디바이스 드라이버가 아니라, SCSI 디바이스 드라이버 계층과 같은 어 떤 서브 시스템을 위한 유사 디바이스 드라이버이기도 한다. 장치 파일은 장치의 유형을 구 별하는 메이저 번호와, 한 덩어리 또는 그 메이저 유형의 한 사례를 구별하기 위한 마이너 유형으로 참조한다. 예를 들어, 첫번째 시스템에서 IDE 컨트롤러에 있는 IDE 디스크들은 메 이저 번호로 3을 가지며, IDE 디스크의 첫번째 파티션은 마이너 번호로 1을 가진다. 따라서 ls -l /dev/hda1을 하면 다음과 같은 출력을 보여준다.

$ brw-rw---- 1 root disk 3, 1 Nov 24 15:09 /dev/hda1

커널에서, 모든 장치는 kdev_t 자료형으로 유일하게 표현된다. 이는 2바이트 길이로 첫번 째 바이트는 마이너 장치 번호를, 두번째 바이트는 메이저 장치 번호를 갖는다[7]. 위에 보여 준 IDE 장치는 커널에서 0x0301을 갖는다. 블럭 장치나 문자 장치를 나타내는 EXT2 inode는 장치의 메이저 번호와 마이너 번호를 첫번째 직접 블럭 포인터(direct block pointer)에 가지고 있다. VFS가 이를 읽으면, 이를 나타내는 VFS inode 자료구조는 이것의 i_rdev 항목을 올바 른 장치 식별자로 설정한다.

번역 : 고양우, 심마로, 이호, 김기용, 서창배, 이대현
정리 : 고양우, 이호

---

1) 음, 고의는 아니었겠지만, 나는 리눅스가 가진 개발자보다 많은 변호사를 가진 운영체제 에 물려왔다.

2) "maximal mount count reached, running e2fsck is recommended"

역주 3) 리눅스를 기본으로 설치하면 블럭 크기는 512바이트이다. (심마로)

역주 4) 1대1 관계이다. (심마로)

역주 5) chmod가 변경하는 항목이 이것이다. (심마로)

역주 6) 디바이스 드라이버를 포함하여 다른 커널 부분도 proc_register_dynamic() 함 수에 적절한 인자를 전달하고, 파일 연산을 수행할 수 있는 함수를 구현함으로써 proc 파일 시스템에 엔트리를 만들 수 있다. (flyduck)

역주 7) 유닉스에서 전통적으로 장치의 번호를 간직하는데 dev_t라는 자료형을 사용하며, 이는 16비트 정수로 메이저 번호와 마이너 번호로 각각 8비트씩 갖는다. 그러나 이는 256개씩의 메이저 번호와 마이너 번호밖에 가질 수 없어서 문제를 가지는데, 그렇다고 이 자료형을 바꾸는 것은 장치번호가 16비트라고 가정하고 있는 소프트웨어에서 문제를 잃으킬 수 있다. 그래서 리눅스는 장치 번호를 나타내는데 kdev_t라는 새로운 자료형을 선언하고 이를 사용하고 있다. 이 자료형은 메이저 번호와 마이너 번호가 각각 16비트의 크기를 갖는다. include/linux/kdev_t.h 참조 (flyduck)