Large Disk HOWTO

Andries Brouwer, aeb@cwi.nl v2.0, 22 January 1999

김남규 nkkim@ricl9.konkuk.ac.kr 1999년 7월 10일

디스크 구조정보(geometry)와 1024개의 실린더 제한에 대하여

제 1 절 문제점

당신이 1024개 이상의 실린더를 갖는 디스크를 가지고 있다고 생각해 보자. 또한 디스크 입출력을 위해 구식의 INT13 BIOS 인터페이스를 사용하는 운영체제를 가지고 있다고 생각해 보자. 이와 같은 경우 당신은 문제점을 겪게 된다. 왜냐하면 이와 같은 인터페이스는 입출력 수행을 위해 10비트를 실린더 수에 할당하여 사용하기 때문이다. 그러므로 실린더 번호가 1024를 넘는 경우 이에 대한 접근이 불가능하다. (2^10 = 1024)

다행스럽게도 리눅스는 BIOS에 의존하지 않으므로 다음 두가지를 제외하면 이러한 문제는 없다.

첫째로 여러분이 시스템을 부팅할때, 리눅스는 아직 작동되지 않은 상태이므로, BIOS문제로 부터 여러분을 해결해 주지 못한다. 이것은 LILO(LInux LOader)와 유사한 종류의 부팅 로더에게 중요한 영향을 미친다.

둘째로 이것은 파티션의 위치를 일치된 하나의 디스크에서 얻어내는 모든 운영체제에 필수적이다. 달리 말해서, 여러분이 리눅스와 도스를 하나의 디스크에서 사용한다면, 두 운영체제는 동일한 방법으로 파티션 테이블을 해석해야 한다. 이것은 리눅스 커널과 `fdisk` 프로그램에 매우 중요한 영향을 밉힌다.

아래에 관련된 모든 내용이 자세히 기술된다. 저자는 커널 버전 2.0.8을 기준으로 설명한다. 다른 버전은 약간의 차이가 있을 것이다.

제 2 절 요 약

대용량 디스크를 가지고 있습니다. 무엇을 해야 할까요 ? 소프트웨어적인 작업으로는 파티션을 생성하기 위해 `fdisk` 나 `cfdisk` 를 실행시키는 것이 우선이겠죠. 그리고 파일시스템을 만들기 위해 `mke2fs` 를 실행하구요. 마지막으로 새롭게 만든 파일 시스템을 거다란 파일 계층 구조 위에 `mount` 시키는 일이죠.

솔직히 말해 최근들어 대용량 디스크로 인한 문제는 없기 때문에 이 글을 읽지 않으셔도 됩니다. 문제점의 대부분은 문제가 있을 거라고 생각하는 사람들 때문이거나 디스크 관리자를 설치하므로서 생기는 경우 또는 `fdisk` 전문가 모드에서 작업하거나 아니면 디스크의 구조적인 정보를 LILO 또는 커널 명령어 라인상에 적는 등의 일로 부터 시작된다.

전형적인 문제점들은 (i) 구식의 하드웨어나, (ii) 동일한 디스크상에 위치한 다중 운영체제나, (iii) 가끔씩 부팅에 관한 것이다.

충고 :

대형 SCSI 디스크는 리눅스의 초기부터 지원되어 왔다. 추가적인 작업이 필요없다. 대형 IDE 디스크는 최근의 안정된 커널(2.0.34 이상)을 받기 바란다. 일반적으로 모든 것이 현재는 정상적일 것이다. 만약 부팅시 LILO 단계에서 멈추는 경우, `/etc/lilo.conf` 에 위치한 구성파일에 `linear` 옵션을 직접 적어주기 바란다.

여러분이 구버전의 `fdisk` 프로그램을 가지고 있고, 이 프로그램이 6(overlapping) 파티션 이라는 경고메세지를 나타내면 이러한 경고 메세지를 무시해 버리거나 `cfdisk` 를 사용하여 실제로 정상적으로 설정되었는지 확인하여라.

만약 디스크 크기에 문제가 있다고 생각되면, 이진수와 십진수 단위(3())를 혼동하고 있지 않는지 확인하기 바라며, 빈 디스크에 대해 `df(disk free)` 명령이 나타내는 빈공간에 대한 크기가 파티션 크기 보다 다소 작다는

것을 기억하기 바란다. 이것은 관리 측면의 부가적인 부분이다. (ext2 파일 시스템은 파일 시스템을 생성하는 단계에서 전체 파티션의 크기에서 일부를 관리 측면(슈퍼유저)에서 예약해 놓는다.)

여전히 문제가 있다고 생각되거나, 호기심이 발동되는 경우, 계속 읽어 주기 바란다.

제 3 절 단위와 크기

1 kilobyte (kB) 는 1000 바이트이다. 1 megabyte (MB) 는 1000 kB이고, 1 gigabyte (GB) 는 1000 MB이다. 1 terabyte (TB) 는 1000 GB이다. 이것은 SI이다. 어떤 사람들은 1 MB의 크기를 1024000 바이트로 사용하여 1.44 MB 의 플로피 디스크에 대해 얘기하는 사람도 있고, 어떤 사람은 1 MB = 1048576 바이트로 사용하는 사람도 있다.

여기에서 저자는 을 따른다. 그리고 이진 단위를 위해 Ki, Mi, Gi, Ti를 사용한다. 그러므로 플로피 디스크는 1440 KiB 이고, 1 MiB 는 1048576 바이트(1.05 MB)이다. 그리고 1 GiB 는 1073741824 바이트 (1.07 GB)이고, 1 TiB 는 1099511627776 바이트(1.1 TB)이다.

보다 정확하게 얘기하면, 디스크 드라이브 제조업체는 SI 표준을 따르며, 십진 단위를 사용한다. 아무튼 리눅스 부팅시의 메세지나 몇몇 fdisk 프로그램은 MB 또는 GB 의 기호를 이진법 대신 사용하거나 혼합된 이진, 십진 단위를 사용한다. 그러므로 디스크 구입당시 크기 보다 작다고 생각하기 이전에 디스크 크기를 십진단위로 정확하게 계산해 보기 바란다.

3.1 섹터 크기

본 문서에서 섹터의 크기는 512 바이트이다. 이 값은 대부분의 경우 동일하다. 그러나 몇몇 MO disk 들은 2048 바이트 크기의 섹터를 사용한다. 그러므로 아래에서 주어지는 크기 데이터는 4배 곱해져야 한다.

3.2 디스크 크기

C 개의 실린더와, H 개의 헤더, 그리고 트랙당 S 개의 섹터를 갖는 디스크는 총 C*H*S 개의 섹터를 갖는다. 그리고 C*H*S*512 바이트 크기의 저장 공간을 갖는다.

예를 들어 디스크 표면에 C/H/S=4092/16/63 이라고 적혀 있으면, 이 디스크는 4092*16*63 = 4124736 개의 섹터를 갖는다. 그리고 4124736*512=2111864832 바이트 (2.11 GB)크기를 갖는다. 8.4 GB 보다 큰 디스크들에 대해서 C/H/S=16383/16/63으로 표기하는 산업약정이 존재한다. 그러므로 디스크의 크기정보는 더 이상 C/H/S 값으로 부터 구할수 없다.

제 4 절 디스크 접근

디스크상의 데이터를 읽거나 쓰기 위해서는 섹터번호나 블럭번호를 이용하여 디스크상의 위치를 명시해야 한다. 만약 SCSI 디스크의 경우라면, 섹터 번호는 직접 SCSI 명령으로 전달된다. 그리고 디스크에 의해 인식된다. 또한 LBA(Large Block Addressing)를 이용하는 IDE 디스크라면 동일한 방법이 이용된다. 그러나 구형의 RLL 이나 MFM 또는 LBA 이전의 단계의 IDE의 경우, 디스크 하드웨어는 디스크상의 특정 위치를 나타내기 위해 세가지의 정보 (cylinder, head, sector)를 요구한다.

Ø荀汰岵鮫古linear) 번호 매김 방법과 3차원 표기법(CHS) 사이에는 다음과 같이 변환될 수 있다. C 개의 실린더, H 개의 헤드, 트랙당 S 개의 섹터를 갖는 CHS(3D) 표현 방법은 일차적인(linear) 번호 매김 방법이나 LBA 방법으로 표현시 c*H*S + h*S + (s-1) 이 가리키는 위치와 동일하게 된다. (수식 마지막에서 1을 뺀것은 오래전 부터 sector 번호의 경우 0이 아닌 1을 시작값으로 사용했기 때문이다)

결과적으로 매우 오래된 non-SCSI 디스크에 접근하기 위해 C, H, S 와 같은 디스크의 물리적 구조정보(*geometry*)를 알아야 한다.

4.1 BIOS를 이용한 디스크 접근과 1024 실린더 크기 제한

리눅스는 BIOS를 이용하지 않는다. 그러나 다른 몇몇 시스템은 이와 같은 방법을 사용하고 있다. LBA 방법 이전에 나온 BIOS 들은 C/H/S 를 파라메터로 하는 INT13 디스크 입출력 루틴을 제공한다. (보다 정확하게 얘기하면, `AH` 레지스터는 실행할 함수를 지정하고, `CH`는 실린더 번호의 하위 8비트를, `CL`은 7-6 비트에 실린더의 상위 2비트를 포함하며, 0-5 번 비트는 섹터 번호로 `DH` 레지스터는 헤드 번호, `DL` 레지스터는 드라이브 번호를 지정한다. 이후 함수를 실행시키면, C/H/S 정보를 얻을 수 있다)

렝肩¯한 방법으로 3 바이트에 CHS 정보를 저장할 수 있다. 10 비트 공간에 실린더 번호를 그리고 8 비트 공간에 헤드 번호, 6비트 공간에 섹터번호를 저장할 수 있다. 이와 같은 방법으로는 실린더 번호의 경우 0-1023 까지만 지정이 가능하다. 1024 번 이상의 실린더는 BIOS에서 접근이 불가능 하다는 결론이다.

도스와 윈도우즈 프로그램은 LBA를 지원하는 IDE 디스크가 나왔을 때, 이전의 방식을 고수하고 있었다. 그러므로 도스와 윈도우즈는 실제적인 디스크 입출력을 위해 더이상 필요하지 않는 상태에서도, BIOS를 이용하기 위해 디스크의 물리적 구조정보(Geometry)를 필요로 하는 상황이 계속되었다. 이것은 다시 말해 최근의 디스크일지라도 BIOS나 다른 운영체제와의 통신시이 요구되는 경우 리눅스는 디스크의 물리적 구조정보(Geometry)를 필요로 한다는 것이다.

이와 같은 문제는 4년정도 지속되었고, 이후 생산되는 디스크는 INT13 시스템 함수에 의한 주소 지정을 할 수 없게 되었다. ((C,H,S)를 위한 10+8+6=24 bits는 8.5 GB 이상의 주소를 지정할 수 없기 때문이다) 이후 새로운 BIOS 인터페이스는 설계되었다: 확장 INT13 시스템 함수라고 불려진 이 방법은 DS:SI 레지스터로 8 바이트의 시작 절대 블럭번호를 포함하는 16 바이트의 디스크 주소단위를 가리킨다.

매우 느린속도로 마이크로 소프트 세계도 이러한 확장 INT13 시스템 함수를 이용하는 방향으로 이동하고 있다. 아마도 지금 부터 몇년안에 디스크의 물리적 구조정보(Geometry)를 필요로 하는 최신의 하드웨어는 더 이상 존재하지 않을 것이다.

4.2 BIOS와 IDE 의 제한에 관한 이야기

ATA 사양 (for IDE disks) - 137 GB 제한

최대 65536 개의 실린더(0-65535), 16 헤드 (0-15), 255 개의 트랙당 섹터 (1-255)로서 최대 267386880 섹터(섹터당 512 byte)를 갖으며, 이것은 136902082560 바이트 크기(137 GB)를 의미한다. 이 사양은 1999년 현 시점에선 문제되지 않는다. 그러나 몇년후면 이 사양도 같은 문제점을 겪게 될 것이다.

BIOS Int 13 - 8.5 GB 제한

최대 1024개의 실린더 (0-1023)와 256개의 헤드 (0-255), 63 개의 트랙당 섹터수(1-63)로서 최대 8455716864 바이트의 크기가 가능하다. 이 사양은 현재 심각한 문제다. 이 방법은 DOS에서 최근의 대용량 디스크를 사용할 수 없다는 것을 의미한다.

528 MB 제한

동일한 C,H,S값이 BIOS INT 13 시스템 호출과 IDE 디스크 입출력에 사용되는 경우, 두가지 제한이 모두 적용된다. 그러므로 1024 개의 실린더와 16개의 헤드, 63 개의 트랙당 섹터로 인해 전체 용량이 528482304 바이트 (528 MB)로 제한되는 문제가 있고, 구형 BIOS를 사용하는 DOS에서는 504 MiB 크기 제한이라는 유명한 문제가 있다. 이러한 문제점은 1993년쯤 부터 시작되었고, 사람들은 모두 LBA와 같은 하드웨어나 BIOS 수준의 변환과 같이 firmware 수준의 편법이나 disk 관리자와 같은 소프트웨어 수준의 편법에 의존하였다. '변환'이라는 개념은 1994년에 나타났다: BIOS 는 드라이브와 통신하는 동안 디스크의 물리적 구조정보(Geometry)를 사용할 수 있고, 다른 방법으로 BIOS가 DOS와 통신하는 경우 구조정보를 속이는 방법으로, 이 둘 사이에 변환을 한다.

2.1 GB 제한 (1996. 4월)

몇몇의 구형 BIOS 들은 CMOS RAM에서 12 비트만을 실린더 수를 위해 할당하였다. 결과적으로 이 수치는 최대 4095 크기를 지정할수 있고, 이것은 다시 4095*16*63*512=2113413120 바이트만을 접근할 수 있다. 대용량의 디스크를 사용하는 경우 부팅시 동작이 멈추게 될 것이다. 이것은 4092/16/63의 구조정

보를 갖는 디스크를 기억속에 남게 하였다. 여전히 대용량 디스크 드라이브들은 4092/16/63개의 디스크의 물리적 구조정보(Geometry)를 설정할 수 있도록 하는 점퍼를 가지고 출시된다. over2gb.html 을 참고하기 바란다.

3.2 GB 제한

Phoenix 4.03 과 4.04 BIOS firmware에는 3277 MB 이상의 드라이브를 갖는 시스템에서 CMOS 설정시 멈춰버리는 문제의 버그가 있었다. See over3gb.htm 을 참고하기 바란다.

4.2 GB 제한 (1997. 2월)

간단한 BIOS 변환 작업은 실린더의 수가 1024를 넘지 않는 범위에서 반복적으로 헤드의 수를 배수로 하고, DOS에서 인식되는 실린더의 수를 반으로 감소시키므로 이루어진다. (ECHS=Extended CHS, 때로는 '대용량 디스크 지원' 또는 그냥 '대형'이라고 불려진다.) DOS와 윈도우 95는 256 개의 헤드를 처리할 수 없고, 일반적인 경우 디스크는 16개의 헤드를 갖는 것처럼 할 것이다. 그러므로 이러한 단순한 방법은 단지 8192*16*63*512=4227858432 바이트 까지만 처리가 가능하다. (1024개의 실린더, 128개의 헤드, 트랙당 63개의 섹터를 갖는 것처럼 변환시키는 방법으로) 확장(Extended) CHS 방법은 트랙당 섹터의 수를 변경하지 않는다. 그러므로, 섹터의 수가 63이 아니라면, 그 제한값은 더 낮을 것이다. over4gb.htm 을 참고하기 바란다.

7.9 GB 제한

좀더 잘 설계된 BIOS들은 먼저 헤드의 수를 15로 조정하므로서 240 개의 헤드를 갖는 위조된 구조정보를 얻음으로서 앞에서의 문제점을 피해 나간다. 1024*240*63*512=7927234560 bytes의 결과를 얻을 수 있다.

8.4 GB 제한

마지막으로 BIOS가 이러한 변환작업을 성공적으로 만들기 위해 할수 있는 모든 방법을 행하고 255개의 헤드와 63개의 트랙당 섹터수를 사용하면 이것은 1024*255*63*512=8422686720 바이트 까지 접근 가능하다. 이 수치는 256개의 헤드를 갖는 디스크의 물리적 구조정보를 사용할 수 없는 제한으로 인해 이전의 8.5 GB 제한보다 근소하게 작은 수치가 된다. (이 변환은 전체 디스크의 용량을 1024*H*63*512에 맞추기 위해 H의 수치를 16, 32, 64,128, 255중 하나를 선택하여 사용한다. 그런 다음 전체용량을 H*63*512으로 나눠 실린더의 수를 계산한다.)

이 같은 주제에 대한 다른 토론자료를 원하면 BreakingtheBarriers라는 것을 참고하라. 그리고 보다 자세한 것을 원한다면 IDEHardDiskCapacityBarriers를 참고하기 바란다.

8.4 GB이상의 하드 디스크는 이러한 디스크의 구조정보를 16383/16/63 으로 알려준다. 이것은 결과적으로 디스크의 물리적 구조정보(Geometry)를 진부하게 만들었음을 의미한다. 그리고 전체 디스크 크기는 더이상 디스크의 물리적 구조정보(Geometry)로 부터 계산할 수 없음을 의미한다.

제 5 절 부팅

시스템 부팅시, BIOS는 첫번째 디스크(또는 플로피 또는 cdrom)의 0 번째 섹터(MBR : Master Boot Record)를 읽는다. 그리고 그곳에 위치한 코드 부분으로 이동을 한다. 일반적으로 그곳에 bootstrap 로더가 위치해 있다. 이 작은 bootstrap 프로그램은 특별한 드라이버 없이 BIOS 의 서비스를 이용한다. 그러므로 리눅스 커널이 1024 실린더 내에 전부 위치하면 부팅이 가능하다는 것을 의미한다.

이로 인한 문제는 매우 쉽게 해결된다: 먼저 커널이나 LILO map 파일과 같이 bootup 시에 사용되는 파일이 BIOS에서 접근할 수 있도록 파티션 전체가 1024 실린더 내에 포함되는지 확인 하라. (아마도 첫번째 또는 두번째 디스크가 이에 해당 할 것이다.)

그러므로 10 MB 정도의 작은 크기의 파티션을 생성하여, 커널이 다루기 쉬운 공간이 되도록 한다. 그리고 첫번째 또는 두번째 디스크의 1024 실린더 안쪽에 커널이 모두 위치할 수 있는지 확인하기 바란다. 다음 /boot 디렉토리 이하에 생성된 파티션을 마운트하여 LILO 가 나머지 필요한 파일을 복사할 수 있도록 한다.

boot loader와 BIOS가 디스크의 구조정보(Geometry)에 동의하여야 한다. 이것은 LILO에 일차적인(linear) 옵션을 주는데 도움이 될 것이다. 자세한 내용은 아래에 있다.

제 6 절 디스크 구조정보, 파티션, 그리고 '중복(overlapping)'

만약 여러분이 디스크상에 여러 종류의 운영체제를 가지고 있다면, 각각의 운영체제는 하나 또는 여러개의 디스크 파티션을 사용한다. 이러한 파티션 위치에 대한 차이는 끔찍한 결과를 가져올 수 있다. MBR(Master Boot Record)은 Primary 파티션이 어느 위치에 존재하는지에 대한 *파티션 테이블*을 가지고 있다. MBR에는 4개의 Primary 파티션 테이블 항목이 있다.

다음과 같은 구조를 갖는다.

```
struct partition {
        char active;    /* 0x80: bootable, 0: not bootable */
        char begin[3];  /* CHS for first sector */
        char type;
        char end[3];    /* CHS for last sector */
        int start;      /* 32 bit sector number (counting from 0) */
        int length;     /* 32 bit number of sectors */
};
```

(위의 구조체는 각각 16 바이트의 크기를 갖는다. 512 Byte 의 MBR에서 후반부에 16 x 4 (Primary Partition Table) 의 공간이 이를 위하여 설정된다)

앞에서도 언급했듯이 위에서 CHS 는 Cylinder, Head, Sector를 의미한다. 이 정보는 중복되는 정보로서 파티션의 위치정보는 24 비트 크기의 시작(`begin`)과 끝(`end`)을 나타내는 필드와 32 비트의 시작(`start`)과 끝(`length`) 필드에 의하여 각각 지정될 수 있다. (위에서 (begin, end) 와 (start, length)에 의해 모두 각각 표현될 수 있다)

리눅스는 단지 시작과 길이 필드만을 사용한다. 그러므로 2^32 보다 작은 크기의 섹터에 대해서만 처리가 가능하다. 즉, 최대 2 TiB 크기의 파티션이 가능하다. 이 크기는 현재 가능한 디스크 크기에 약 100 배 정도 큰 크기이다. 그러므로 앞으로 8 년이상은 충분할 것으로 예상된다. (그러므로 파티션은 크기가 커질수 없다. 그러나 ext2 파일 시스템상의 파일에는 심각한 제한이 있다. 그것은 32 비트 정수형을 갖는 하드웨어상에서의 ext2 파일시스템에서는 2 GiB 이상 참조할 수 없다는 것이다.)

DOS 는 시작과 끝 필드를 사용한다. 그리고 디스크를 접근하기 위해 BIOS INT13 시스템 호출을 사용한다. 그러므로 변환하는 BIOS 에서 조차도 8.4 GB 보다 작은 공간만을 접근할 수 있다. 파티션의 크기는 FAT16 파일 시스템의 제한으로 인해 2.1 GB 보다 커질 수 없다뉜활갸별職李Tel⊃indows 3.11 과 WfWG 와 windows NT 3.*, Novell NetWare에서 발생한다.

Windows 95는 확장 INT13 인터페이스를 제공해 왔다. 그리고 특별한 파티션 유형을 사용해 왔다.(b, 6, 5 대신 c, e, f를 사용하는 것과 같이) 위와 같은 유형을 지정하므로서 파티션을 참조시 확장 INT13 방법으로 참조할 수 있도록 하고 있다. 이러한 파티션 유형이 사용될 경우, 시작(`start field`)과 끝(`end field`)을 나타내는 필드는 1023/255/63과 같은 무의미한 정보를 갖는다 Windows 95 OSR2 는 FAT32 파일 시스템을 소개하였고, 이 파일 시스템의 유형은 b 또는 c 를 갖는다. 또한 이 파일 시스템은 2 TiB 크기 까지의 파티션 크기를 지원한다.

`fdisk`에서 파티션 '중복(overlapping)' 메세지를 보게 되는데 어떤 의미인가요 ? 문제가 없나요 ?

문제가 있습니다. 이러한 파티션의 시작(`start`) 과 끝(end) 필드를 보면 DOS 운영체제에서와 같이 중복되어 있는 것을 볼 수 있습니다. (또한 수정되지 않습니다. 왜냐하면 이러한 필드들은 1024 개 이상의 실린더 번호를 기록할 수 없기 때문입니다. - 1024 개 이상의 실린더를 갖는 이상 '중복' 현상은 지속될 것입니다.)

어쨌거나, 여러분이 시작(`start`)과 끝(end) 필드를 보게되면 리눅스나 Windows 95 가 c, e, f 유형의 파티션 유형에 대해 그러하듯이 정상적입니다. 그러므로 `cfdisk` 가 정상적이고, 여러분이 Linux만을 사용하는 디스크를 가지고 있다면 이러한 경고 메세지를 무시하기 바랍니다. 그러나 이러한 디스크가 DOS 와 공유될 경우

조심하기 바랍니다. /dev/hdx 에 있는 파티션 테이블을 확인하려면 cfdisk -Ps /dev/hdx 와 cfdisk /Pt /dev/hdx 같은 명령을 사용하기 바랍니다.

제 7 절 변환 및 디스크 관리

디스크의 헤드, 실린더, 트랙과 같은 구조정보는 MFM이나 RLL 형태의 디스크 시대부터 존재하던 것이다. 그 시대에 이러한 구조정보는 물리적인 정보와 동일하였다. 근래에 IDE 나 SCSI 디스크에서는 그 누구도 실제적인 디스크의 구조정보가 어떻게 되는지 관심을 두지 않는다. 실제로 트랙당 섹터의 수는 변동적이다 - 예를 들어 디스크의 바깥쪽 트랙에 할당된 섹터의 수가 안쪽 트랙보다 많다. 그러므로 트랙당 섹터의 정확한 수치는 존재하지 않는다. 다소 떨어진 얘기로 IDE INITIALIZE DRIVE PARAMETERS (91h) 명령은 현재 디스크가 가지고 있는 것으로 가정되는 헤드와 트랙당 섹터의 수를 디스크에 알려준다. 2개의 헤드를 갖는 대형의 최신 디스크가 15 또는 16개의 헤드를 가지고 있다고 BIOS 에게 알리는 것은 꽤 일반적인 일이며, 이 같은 경우 BIOS는 다시 사용자 프로그램에 255개의 헤드가 있는 것으로 알린다.

사용자에게는 디스크를 마치 섹터 번호에 의해 참조될 수 있도록 0, 1, 2,..의 일차원적인 배열로 간주하는 것이 적격이다. 그리고 특정 섹터의 디스크상 위치 정보는 firmware 수준에서 처리되도록 하는 것이 적격이다. 이와 같은 일차적 번호부여 방법을 LBA 라고 한다.

그러므로 개념적인 그림은 다음과 같다. DOS, 또는 몇몇의 boot loader는 BIOS 와의 통신을 위해 (c,h,s) 방법을 사용한다. BIOS 는 (c,h,s) 방식을 구조정보(Geometry) 속임기법을 사용하여 사용자가 사용하는 LBA 방식으로 변환한다. 디스크가 LBA를 받아 들이면, 이 정보는 디스크 입출력을 위해 사용된다. 그러나 그렇지 않으면, 현재의 디스크가 사용하는 구조정보를 이용하여 (c',h',s') 방식으로 역변환되고, 이를 이용하여 디스크 입출력이 이뤄진다.

언어사용에 있어 혼돈이 있으니 주목하라 : : 'LBA'는 디스크 용량을 표시하는 용어로서 이것은 'Linear Block Addressing' 을 의미하고(CHS 주소방법에 반대되는 방법), BIOS 구성에서 이것은 'assisted LBA' 라고 불려지는 변환기법을 의미한다. 아래'4.2()'를 보기 바란다.

firmware가 LBA를 지원하지 않지만 BIOS가 변환에 대해 알고 있는 경우, 유사한 작업이 일어난다. (구성단계에서 이것은 'Large'라고 표시된다) 이제 BIOS 는 (C,H,S)의 구조정보를 운영체제에게 제공하고, 디스크 컨트롤러와 (C',H',S') 정보를 이용한다. 일반적으로 S = S', C = C'/N, H = H'*N 에서 N은 C' <= 1024를 만족시키는 2의 지수승으로 가장 작은 값이다. 그러므로 C' = C/N에서 버려지는 용량을 최소화 한다. 다시 이것은 8.4 GB (7.8 GiB)까지 엑세스를 가능케 한다.

(세번째 설치 옵션은 일반적으로 'Normal', 이것은 아무런 변환 과정도 필요 하지 않는 방법이다.)

BIOS에서 'Large' or 'LBA' 모드를 지원하지 않아도, 소프트웨어적인 해결책이 있다. OnTrack 이나 EZ-Drive같은 디스크 관리 프로그램은 BIOS의 디스크 핸들링 루틴을 그들의 루틴으로 대체시킨다. 때때로 이것은 MBR과 이후의 sector에 디스크관리용 코드를 위치시키므로서 수행된다. (OnTrack 프로그램에서는 이 코드를 DDO(Dynamic Drive Overlay) 라고 부른다.) 그래서 이것은 다른 운영 체제이전에 부트된다.

이것이 바로 Disk 매니저가 설치된 상태에서 플로피로부터 부팅시 겪게 되는 문제이다.

결과는 BIOS를 통한 변환과 거의 비슷하다. 그러나 동일한 디스크상에서 여러가지의 운영체제를 사용하는 경우, 디스크 매니저는 더 많은 문제점을 야기 시킬수 있다.

Linux는 1.3.14 이후 버전부터 OnTrack Disk Manager 를 지원해 왔고, EZ-Drive는 1.3.29 이후 버전 부터 지원을 하고 있다. 보다 자세한 내용은 아래에 적혀 있다.

제 8 절 IDE 디스크를 위한 커널의 디스크 변환(?)

만약 리눅스 커널이 IDE 디스크에 설치된 디스크 매니저를 감지하는 경우 리눅스 커널은 디스크 매니저가 행한 방법과 동일한 방법으로 디스크를 다시 배치하려고 할 것이다. 그러므로 Linux는 DOS와 같이 동일한 디스

크 파티션을 인식하게 될 것이다. 명령 라인에서 구조정보가 입력되는한 재배치 작업은 일어나지 않는다. 그러므로 'hd=*cyls*,*heads*,*secs*' 와 같은 명령라인 옵션은 디스크 매니저와의 호환성을 떨어 뜨리게 될 것이다.

재배치 과정은 C <= 1024 이거나 H = 255가 될 때까지 H*C값을 일정하게 유지한체로 헤드의 수를 4, 8, 16, 32, 64, 128, 255 로 차례로 시도함으로서 이루어 진다.

자세한 내용은 아래에 있다. - 아래의 부제목은 해당 부팅 메세지로 나타나는 문자열이다. 파티션의 유형은 항상 16진수로 표시된다.

8.1 EZD

EZ-Drive 는 프라이머리 파티션의 유형이 55값을 갖으며, 이를 이용하여 인식할 수 있다. 구조정보는 위에서 언급된 것처럼 재배치 된다. 그리고 sector 0에서 부터의 파티션 테이블은 무시된다. 대신에 파티션 테이블은 sector 1에서 부터 읽어 진다. 디스크 블럭 번호는 변경되지 않는다. 그러나 sector 0 번에 쓰는 작업은 sector 1에 쓰도록 변경된다. 이러한 작업은 kernel에서 아래와 같이 변경후 커널을 다시 컴파일 함으로서 이루어진다.

`#define FAKE_FDISK_FOR_EZDRIVE 0` in `ide.c`.

8.2 DM6:DDO

OnTrack DiskManager (첫번째 디스크상에 있는)는 첫번째 프라이머리 파티션의 유형이 54값을 갖으며, 이를 통해 인식 가능하다. 구조정보는 위에서 언급된 방법으로 재배치된다. 그리고 전체 디스크는 63개의 sector만큼 쉬프트 된다. 그러므로 이전의 63번 섹터는 0번 섹터가 된다. 결국 새로운 MBR(파티션 테이블을 포함)은 새로운 섹터 0번으로 부터 읽혀진다. 물론 이러한 쉬프트 작업은 DDO를 위한 공간을 남겨 둘것이다. 이것이 다른 디스크에 대해서는 쉬프트 작업을 하지 않는 이유이다.

8.3 DM6:AUX

OnTrack DiskManager (다른 디스크에 설치된 경우)는 첫번째 프라이머리 파티션의 유형이 51또는 53을 갖기 때문에, 이를 이용하여 식별이 가능하다. 구조정보는 위에서 언급된 것처럼 재 배치 된다.

8.4 DM6:MBR

OnTrack DiskManager의 구버전은 파티션 유형정보를 이용하지 않고, 기호를 이용하여 식별한다. (MBR의 2번째와 3번째 바이트에서 찾은 offset값이 430보다 작은지를 확인하고 이 옵셋에서 발견된 short형의 값이 0x55AA와 같은지 확인한다. 그리고 홀수 바이트가 뒤를 따르는지 확인한다. 구조정보는 위와 동일하게 변환된다.

8.5 PTBL

마지막으로, 프라이머리 파티션의 시작과 끝 값으로 부터 변환을 유추하려는 시도가 있다. 만약 어떤 파티션이 시작(`start`)과 끝(`end`) 섹터 번호로 1과 63을 각각 갖고, 마지막 헤드번호로서 31, 63, 127 또는 254를 갖는 것은, 파티션을 실린더와 경계상에서 끝나도록 하는 관례적인 이유와 IDE 인터페이스가 최대 16개의 헤드를 사용하기 때문이다. 또한 이것은 변환이 이루어졌다고 추측할 수 있고, 구조정보는 32, 64, 128 또는 255개의 헤드로 변환된다. 그렇지만, 현재의 구조가 트랙당 63개의 섹터를 갖고 있고, 적어도 헤드의 수만큼 갖는 경우 재배치 작업은 일어나지 않는다. (이것은 이미 재배치 작업이 일어났음을 의미하기 때문이다.)

제 9 절 결 론

이 모든 것이 의미 하는 것이 무엇인가 ? 리눅스 사용자에게는 단 한가지 : 사용자들은 LILO와 `fdisk`가 올바른 구조정보를 사용하는지 확인해야 한다는 것이다. `fdisk`에서 올바르다는 것은 동일한 디스크상의 다른 운영체제에 의해 구조정보가 사용(공유)되는 것을 의미하며, LILO에서 올바르다는 의미는 부팅시 BIOS와의 성공적인 교신을 할 수 있는 구조정보(Geometry)를 의미한다.

어떻게 `fdisk` 가 구조정보를 알수 있는가 ? `fdisk` 는 `HDIO_GETGEO` ioctl을 이용하여 kernel에게 요구한다. 그러나 사용자는 구조정보를 상호대화식 또는 명령 라인을 통해 덮어 씌울 수 있다.

어떻게 LILO 가 구조정보를 알수 있는가 ? LILO는 `HDIO_GETGEO` ioctl을 이용하여 커널에게 요구한다. 그러나 사용자는 `/etc/lilo.conf` 파일내의 '`disk=`' 옵션을 이용하여 구조정보를 덮어 씌울 수 있다. 어떤 사람은 LILO에게 순차적(`linear`)인 옵션을 줄 것이다. 그리고 이것은 LILO의 map 파일에 CHS 주소 대신 LBA 주소를 저장할 것이다. 그리고 부팅시 구조정보를 찾을 것이다. (디스크 구조정보를 요구하기 위해 INT 13 Function 8 을 사용하는 방법으로)

커널이 무엇을 답해야 할지 어떻게 알 수 있는가 ? 무엇보다도 사용자는 '`hda=`*cyls*`,`*heads*`,`*secs*' 형식의 커널 명령 라인 옵션을 통해 암시적으로 구조정보를 명시했을 것이다. (bootparam(7)을 참조) 그렇지 않았다면 커널은 BIOS나 하드웨어를 통해 얻어진 값을 이용하여 추측할 것이다.

제 10 절 상세한 내용

10.1 IDE 에 관한 상세한 내용 - 7가지의 구조정보(Geometry)

IDE 드라이버는 구조정보를 다섯가지 다른 방법으로 얻을 수 있다. 첫번째(G_user)는 명령 라인상에서 사용자에 의한 명시 방법이다. 두번째(G_bios)는 32 bit 모드로 전환 이전, 시스템 시작시에 읽혀지는 첫째, 두번째 디스크에 대한 BIOS상의 고정된 디스크 항목 테이블이다. 세번째(G_phys)와 네번째(G_log)는 IDENTIFY 명령에 대해 IDE 컨트롤러가 의해서 반환하는 것으로, 이것은 '물리적'이거나 '현재의 논리적' 구조정보이다.

반면에, 드라이버는 구조정보에 관한 두개의 값을 필요로 한다. 한가지는 `HDIO_GETGEO` ioctl에 의해 반환되는 G_fdisk이고, 다른 하나는 실제적인 입출력에 사용되는 G_used이다. G_fdisk 와 G_used 는 G_user 가 주어진 경우, G_user로 초기화 되고, CMOS에 의해 G_bios값이 제공되면 G_bios 값으로 설정된다. 마지막으로 위의 경우에 해당되지 않는 경우 G_phys로 초기화 된다. 만약 G_log가 적당한 값으로 판단되면, G_used는 이 값으로 설정된다. 그렇지 않고 G_used는 부정당해 보이며, G_phys는 적당해 보이면 G_used는 G_phys 값으로 설정된다. 여기에서 적당하다는 것은 헤드의 수가 1-16인 경우를 의미한다.

달리 말하면: 명령 라인 정보가 BIOS 정보를 덮어 씌우고 `fdisk`가 어떤 것을 참조할 지 결정한다. 그러나 만약 이것이 16개 헤드이상을 갖는 변환된 구조정보로 구체화 되면, 커널 입출력에 대해, IDENTIFY 명령의 출력 값에 의해 덮어 씌워진다.

G_bios는 다소 신뢰할 만하지 못하다는 것을 기억하기 바란다. : SCSI 로 부터 부팅하는 시스템의 경우 첫번째나 두번째 디스크는 SCSI 디스크일 것이다. 그리고 sda에 대해 BIOS가 제공하는 구조정보는 커널 내부에서 hda를 위해 사용된다. 게다가 BIOS 설정 단계에서 언급되는 않는 디스크들은 BIOS에 의해 보이지 않는다. 이것은 예를들어 IDE 만을 사용하는 시스템에서 hdb가 셋업 단계에서 주어지지 않은 경우 BIOS 에 의해 제공되는 첫째, 둘째 디스크의 구조정보가 hda와 hdc 에 적용된다는 것을 의미한다.

10.2 SCSI 에 대한 상세한 정보

SCSI의 경우는 약간 차이가 있다. SCSI 명령이 이미 논리적인 블럭 번호를 사용하기 때문에 구조정보(Geometry)는 실제 입출력과 무관하다. 그렇지만 파티션 테이블의 유형은 여전히 동일하다. 그러므로 `fdisk`는 일부 구조정보를 조작해야 하며, `HDIO_GETGEO` 명령을 사용한다. 대신에 `fdisk`는 IDE와 SCSI 디스크사이에 구분을 두지 않는다. 아래의 상세한 정보로 부터 볼수 있듯이 다양한 드라이버들은 각각 조금씩 다른 구조정보를 만들어 낸다. 실제로 하나의 큰 혼합체이다.

만약 여러분이 DOS나 비슷한 유형의 운영체제를 사용하고 있지 않다면, 모든 확장 변환 설정을 피하고 가능하다면, 64개의 헤드와 트랙당 32개의 섹터 (완전하고, 편리한 설정을 위해 실린더당 1 MiB의 크기로, 64*32*512 = 1 MiB)를 사용하여라. 그러므로 디스크를 다른 컨트롤러로 이동시에도 이러한 문제가 발생하지 않도록 하여라. 몇몇 SCSI 디스크 드라이버(aha152x, pas16, ppa, qlogicfas, qlogicisp)는 DOS 호환성에 대해 매우 예민하여 8 GiB이상을 사용하는 리눅스 시스템을 허용하지 않는다. 버그다.

실제 구조정보는 무엇인가 ? 먼저 얘기하면 실제적인(real) 구조정보는 없다는 것이다. 그리고 만약이 있었더라도 여러분은 결코 알고자 하지 않았을 것이고 결코 `fdisk`나 LILO 또는 kernel에 그러한 정보를 전달하지 않을 것이다. 이것은 엄밀히 말해 SCSI 컨트롤러와 디스크간의 사업과 같다. 다시 얘기하면, 단지 어리석은 사용자만이 여전히 `fdisk`/LILO/kernel>에 SCSI 디스크의 실제 구조정보를 전달할 것이다.

그러나 여전히 호기심이 생긴다면, 디스크 자체에 정보를 요구할 수 있다. 전체 디스크의 크기 정보를 반환하는 READ CAPACITY라는 쓸만한 명령이 존재하고 Rigid Disk Driver Geometry Page에서 실린더의 수와 헤드의 수를 알려주는 Format Page에서 섹터당 바이트 크기와 트랙당 섹터의 수를 알려 주는 MODE SENSE 명령이 존재한다.

마지막 번호는 일반적으로 notch에 의존한다. 그리고 트랙당 섹터의 수는 유동적이다. 바깥쪽의 트랙은 안쪽 트랙보다 많은 수의 섹터를 갖는다. 리눅스에서 제공되는 `scsiinfo` 프로그램은 이러한 정보를 제공한다. 이 프로그램에는 어느 누구도 사용하길 원치 않는 복잡하고도 세부적인 내용이 들어 있다. (아마 심지어 운영체제 까지도)

게다가 우리가 `fdisk`나 LILO에 관심을 갖는다면, 파티션 테이블이 C/H/S를 위해 단지 각각 10, 8, 6 비트를 예약해 두었기 때문에 `fdisk`에서는 사용될 수 없는 C/H/S=4476/27/171과 같은 하나의 예를 찾을 수 있다.

그러면 커널의 `HDIO_GETGEO`는 이러한 정보를 어디에서 얻는가 ? 아마도, SCSI 컨트롤러나 입력된 논리에 의한 추측을 통해 얻을 것이다. 몇몇 드라이버는 우리가 실제값을 알고자 하는 것으로 간주하는 듯 싶지만, 우리는 단지 DOS 나 OS/2 FDISK(또는 Adaptec AFDISK나 기타 등등)가 무엇을 사용할 것인지 알고자 한다.

리눅스의 `fdisk`프로그램은 LBA 섹터번호를 c/h/s 주소로 변경하기 위해 트랙당 헤더(H)와 섹터의 수(S)를 필요로 한다는 것을 기억하라. 그러나 실린더의 수(C)는 위의 변환 과정에서 아무런 역할도 하지 않는다. 몇몇 드라이버들은 드라이브의 크기가 적어도 1023*255*63 sector 이상 이라는 것을 알리기 위해 (C,H,S) = (1023,255,63)을 사용한다. 이러한 방법은 실제 크기를 밝히지 않기 때문에 유감스러운 일이다. 그리고 대부분의 `fdisk` 프로그램은 사용자들을 약 8GiB 의 크기로 제한시킨다.

아래의 기술에서, M 은 전체 디스크 공간을 표시하며, C, H, S는 실린더, 헤드 그리고 트랙당 섹터의 수를 나타낸다. C 값을 M / (H*S)로 정의하면, H, S 값을 할당하기에 충분하다.

기본적으로 H = 64, S = 32이다.

aha1740, dtc, g_NCR5380, t128, wd7000:

H=64, S=32.

aha152x, pas16, ppa, qlogicfas, qlogicisp:

C <= 1024 이고 H=64, S=32이면, H=255, S=63이고 C 값은 1023과 M/(H*S)에서 작은 값을 택한다. (그러므로 C 값은 절삭되고, H*S*C는 디스크 크기 M에 대한 근사값이 아니다. 이것은 대부분의 `fdisk` 프로그램을 혼돈스럽게 한다.) `ppa.c` 코드는 M 대신에 M+1값을 사용하고, `sd.c`에서의 버그 때문에 M 값을 1 만큼 크게 설정한다고 얘기한다.

advansys:

C <= 1024 이고 H=64, S=32 이며, BIOS 상에서 '> 1 GB' 옵션이 설정되면, H=255, S=63이 된다.

aha1542:

컨트롤러에서 두가지 가능한 변환방법중 어떤 방법을 사용하고 있는지 요구한다. 그리고 H=255, S=63으로 설정하거나 H=64, S=32로 사용한다. 전자의 경우, "aha1542.c: Using extended bios translation" 이라는 부팅 메세지가 나타난다.

aic7xxx:

C <= 1024, H=64, S=32이고, ”extended” 부팅 파라메터가 주어지거나, SEEPROM 이나 BIOS에 ‘extended’ 비트가 설정되는 경우 H=255, S=63을 갖는다.

buslogic:

C < 1024, H=64, S=32, C < 1024이고, 컨트롤러상에서 확장 변환이 설정된 경우, M < 2^22 이면 H=128, S=32, 그렇지 않은 경우에는 H=255, S=63이 설정된다. (C,H,S)를 결정을 한 후 파티션 테이블은 읽혀지고 다음 세가지의 가능한 경우 중 (H,S) = (64,32), (128,32), (255,63) endH=H-1를 만족하는 한가지 값이 사용된다. 그리고 부팅 메세지는 ”Adopting Geometry from Partition Table” 이라고 출력된다.

fdomain:

BIOS의 드라이브 파라메터 테이블에서 디스크 구조정보를 찾거나, 파티션 테이블을 읽어 들여 첫번째 파티션에 대해 H=endH+1, S=endS을 사용한다. 또는 M < 2^21(1 GiB)의 경우 H=64, S=32를 사용하고, M < 63*2^17 (3.9 GiB)의 경우 H=128, S=63 를 사용하고, 나머지의 경우에 H=255, S=63를 사용한다.

in2000:

C <= 1024를 만족하는 첫번째 쌍을 다음에서 찾아서 사용한다. (H,S) = (64,32), (64,63), (128,63), (255,63) 마지막 경우, 1023으로 절단된다.

seagate:

디스크로 부터 C,H,S 값을 읽어 들인다. (소름끼는 일이죠!) 만약 C 또는 S 가 매우 큰 경우, S=17, H=2로 설정하고 C <= 1024 일때 까지 H를 배수로 만든다. 이것은 M > 128*1024*17(1.1 GiB)인 경우 H를 0으로 설정하게 된다. 이것은 버그다.

ultrastor and u14_34f:

다음의 세가지 경우중 하나가 컨트롤러의 맵핑 모드에 따라 사용된다. ((H,S) = (16,63), (64,32), (64,63))

만약 드라이브 구조정보가 명시하지 않으면, 파티션 테이블이나 전체 디스크 용량을 사용하여 추측하는 방법을 이용하게 된다.

파티션 테이블을 보아라. 실린더 경계상에서 파티션을 끝내는 약정으로 인해 주어진 `end=(endC,endH,endS)`로 부터 H=`endH`+1, S=`endS`로 설정할 수 있다. (섹터 번호가 1번부터 시작하는 것을 기억하기 바란다.)

다음단계는 보다 정확하게 이루어진다. 만약 사용중인 파티션이 존재하면, 최대의 `beginC` 값을 갖는 파티션을 선택한다. 선택된 파티션에 대해 `end+1` 값을 확인한다. 시작(`start`) 위치에 길이(`length`)를 더하여 계산하고 이 파티션이 실린더의 경계에서 끝난다고 가정하므로서 계산된다.

두개의 값이 일치하거나 `endC` = 1023 이고 `start+length` 값이 `(endH+1)*ensS`의 정수배이면 이 파티션은 실제로 실린더 경계상에 맞춰졌다고 가정한다. 그리고 H 에 `endH+1` 을 S 에 `endS` 를 대입한다.

이 과정이 실패하면, 실제로 파티션이 존재하지 않거나, 그들이 상이한 크기값을 갖기 때문이며, 이 경우 디스크의 용량 M 값을 확인하라.

Algorithm: H = M/(62*1024) (rounded up), S = M/(1024*H) (rounded up), C = M/(H*S) (rounded down). 이것은 실린더가 최대 1024값을 갖고 섹터의 경우 62개를 갖는 범위내의 값으로 하는 (C, H, S)를 만들어 내는 결과를 갖는다.

제 11 절 리눅스 IDE 컨트롤러의 8 GiB 제한(The Linux IDE 8 GiB limit)

리눅스의 IDE 드라이버는 디스크의 구조정보와 용량 및 기타 정보를 ATA IDENTIFY 명령(?)을 이용하여 얻는다. 최근까지 드라이버는 C*H*S에 의해 계산된 용량이 반환된 lba_capacity 값보다 10% 이상 큰 경우 이 값

을 신뢰하지 않았다.

어쨌거나 업계간의 약정에 의해, 16514064 섹터보다 큰 용량을 갖는 대용량 IDE 디스크는 그들의 실제 크기에 무관하게 7.8 GB 용량을 나타내는 C=16383, H=16, S=63 값을 반환한다. 그러나 그들의 실제 크기는 lba_capacity를 통해 전달한다.

최근의 리눅스 커널(2.0.34, 2.1.90)은 이러한 내용이 반영되어 있어 정상적으로 처리한다. 만약 여러분이 구버전의 커널을 가지고 있으며, 업그레이드를 원하지 않는다면, 그리고 현재의 커널이 전체 대용량 디스크의 크기중 8 GiB 만을 인식할 수 있다면 `/usr/src/linux/drivers/block/ide.c` 에 있는 `lba_capacity_is_ok` 의 실행부분을 다음과 같이 변경하도록 하여라.

```
static int lba_capacity_is_ok (struct hd_driveid *id) {
        id->cyls = id->lba_capacity / (id->heads * id->sectors);
        return 1;
}
```

보다 신중한 패치 작업을 원한다면 2.1.90를 보아라

11.1 BIOS 복잡성(complications)

언급하였듯이, 대용량 디스크는 실제 크기와 무관하게 C=16383, H=16, S=63 값의 구조정보를 반환한다. 반면에 실제 크기는 LBAcapacity을 통해 반환된다.

몇몇 BIOS 들은 이것을 인식하지 못하고 16383/16/63 값에서 실린더수를 줄이고, 헤드의 수를 늘이는 변환을 한다.

그래서 커널은 단일 구조정보 16383/16/63와 부정확한 값을 반환하는 BIOS의 정보도 인식해야 한다.

커널 2.2.2 이후부터 이러한 문제는 정상적으로 작동된다. (BIOS에서 H 와 S값을 구하고 C 값은 전체용량을 (H*S)로 나누어 구함으로서 해결된다.)

제 12 절 리눅스의 64 GiB 제한

`HDIO_GETGEO` ioctl은 short 형 으로 실린더의 수를 반환한다. 이것은 65535 개 이상의 실린더를 갖는 경우 실린더의 수가 짤려 나갈 수 있다는 것이다. 그래서 80 GiB의 디스크가 16 GiB 크기로 보여질 수 있다는 것이다. 일단 무엇이 문제인지 인식하고 나면 문제는 쉽게 해결 가능하다.

제 13 절 확장 및 논리 파티션

6(위에서) 우리는 MBR(Master Boot Record)(sector 0)의 구조에 대해 보았다. 부트 로더 코드 이후에 위치하는 4개의 파티션 테이블 항목은 각각 16바이트 를 차지하고 있다. 그리고 마지막으로 AA55 값으로 기록되어 있다. 파티션 테이블 항목중 5번이나 F 또는 85(hex) 유형의 파티션은 특별한 의미를 갖는다. 이러한 유형은 확장(*extended*) 파티션을 의미한다. 이것은 여러개의 논리(*logical*) 파티션으로 재 분할됨을 가리키는 표시이다. (그러므로, 확장 파티션은 하나의 큰 상자일 뿐이며, 이것 자체적으로 사용되지 않고 확장 파티션 내부에 논리 파티션을 생성하여 사용한다.) 단지 확장 파티션의 첫번째 섹터에 대한 위치가 중요할 뿐이다. 첫번째 섹터는 4개의 항목을 갖는 파티션 테이블을 포함한다 : 하나의 논리 파티션과 하나의 확장 파티션 그리고 두개의 미사용 파티션으로 구성된다. 이러한 방식으로 하나의 확장 파티션은 디스크상에 흩어져 연결된 파티션 테이블 섹터를 갖는다.

이것을 이해하는 것을 매우 중요하다.: 어떤 사람이 디스크를 파티션 할 때 실수를 한 경우, 데이터가 아직 존재하는 알고 싶을 것이다. 결론은 일반적으로 존재한다는 것이다. 그러나 논리 파티션이 생성되고 나서 논리 파

티션의 정보를 나타내는 파티션 테이블 섹터가 이러한 논리 파티션들의 시작에 기록된 다면 이전에 있던 데이터를 잃어 버리게 된다.

sfdisk 프로그램은 전체적인 연결을 보여 줄 것이다.

```
# sfdisk -l -x /dev/hda

Disk /dev/hda: 16 heads, 63 sectors, 33483 cylinders
Units = cylinders of 516096 bytes, blocks of 1024 bytes, counting from 0

   Device Boot Start     End   #cyls   #blocks   Id  System
/dev/hda1          0+    101    102-    51376+  83  Linux
/dev/hda2        102    2133    2032   1024128  83  Linux
/dev/hda3       2134   33482   31349  15799896   5  Extended
/dev/hda4          0       -       0         0   0  Empty

/dev/hda5       2134+   6197    4064-  2048224+  83  Linux
    -           6198   10261    4064   2048256   5  Extended
    -           2134    2133       0         0   0  Empty
    -           2134    2133       0         0   0  Empty

/dev/hda6       6198+  10261    4064-  2048224+  83  Linux
    -          10262   16357    6096   3072384   5  Extended
    -           6198    6197       0         0   0  Empty
    -           6198    6197       0         0   0  Empty
...
/dev/hda10     30581+  33482    2902-  1462576+  83  Linux
    -          30581   30580       0         0   0  Empty
    -          30581   30580       0         0   0  Empty
    -          30581   30580       0         0   0  Empty

#
```

만약 비정상적인 파티션 테이블을 작성할 수 있다면 확장 파티션이 자신을 다시 가리키거나 파티션 연결의 초기를 가리키도록 된다면 많은 커널이 무한 순환에 빠지게 될 것이다.

두개의 확장 파티션을 이러한 파티션 테이블 섹터중 하나에 만들어서 파티션 테이블 연결을 만드는 것은 가능하다. (이러한 경우는 5, F, 85 를 확장 파티션으로 인식하지 못하는 fdisk 프로그램에서 F 다음 파티션으로 5를 생성하는 경우 발생할 수 있다.) 이러한 상황을 처리할 수 있는 표준적인 fdisk type 프로그램은 없다. 이것을 고치기 위해서는 약간의 수작업이 필요하다.

리눅스 커널은 가장 바깥쪽 레벨에서 포크를 받아 들일 것이다. 다시 말하며, 두개의 논리 파티션 연결을 만들 수 있다. 때로는 이것이 유용하기도 하다. 예를 들어 하나는 5번 유형을 사용할 수 있고, DOS에서 사용된다. 그리고 다른 유형 85는 DOS에서는 사용되지 않으므로 DOS FDISK 는 1024개의 실린터를 지나치는 논리 파티션 때문에 문제시 되지 않을 것이다.

제 14 절 문제 해결

많은 사람들은 그들이 문제를 가지고 있다고 생각한다. 하지만 실제로 그렇지 않다. 또는 그들은 그들이 가지고 있는 문제가 디스크 구조정보 때문이라고 생각한다. 그러나 디스크 구조정보는 이러한 문제와 연관이 없다. 위에 말이 복잡하게 들릴지 모르지만, 디스크의 구조정보 조작은 너무도 쉽다 : 아무 것도 해줄 필요가 없다. 그냥 그대로 모든 것이 정상적이다. 또는 부팅시 'LI' 가 나타나며 더 이상 진행하지 않는 경우 LILO 에서 'linear' 키워드를 주라. 커널의 부팅 메세지를 보아라. 그리고 기억하라.: LILO나 fdisk등에 head나 cylinder를 입력하

는 등의 구조정보를 가지고 시간을 허비하는 일은 하면 할 수록 일이 진행가능성은 줄어 들 것이다. 개략적으로 말하면 모든것이 기본적으로 정상적이다.

그리고 기억하라: 디스크의 구조정보가 사용되는 곳은 리눅스 상에 어디에도 없다. 그러므로 리눅스를 운영하는 동안 디스크 구조정보에 의해 문제를 겪게될 일은 없다. 디스크 구조정보는 단지 LILO와 fdisk에 의해서만 사용된다. 그러므로 LILO가 커널을 부팅하는데 실패하면, 이것은 구조정보 문제인 것이다.

만약 다른 운영체제 시스템이 파티션 테이블을 인식하지 않으면 이것은 구조정보 때문일 것이다. 특별한 다른 이유가 없다. 마우트가 제대로 되지 않더라도 디스크 구조정보에 대해 걱정할 필요가 없다. 문제는 다른 곳에 존재한다.

14.1 문제점: 리눅스가 디스크에 대한 잘못된 구조정보를 만들어 낸다

디스크가 잘못된 구조정보를 갖는 것은 가능하다. 리눅스 커널은 BIOS에게 hd0 와 hd1을 요구한다.(BIOS 상에서 드라이브는 80H와 81H 간주된다) 그리고 이 데이터가 hda와 hdb에 대한 것으로 간주한다. 그러나 SCSI로 부팅하는 시스템에서 처음 두개의 디스크는 아마도 SCSI 디스크가 될 것이다. 그래서 첫번째 IDE 디스크 hda인 5번째 디스크가 sda에게 해당되는 구조정보를 갖게 된다. 이러한 문제는 부팅 파라메터를 다음과 같이 입력 하므로서 해결된다. C, H, S 의 적당한 값 'hda=C,H,S'를 부팅시 또는 /etc/lilo.conf에 설정하므로서 해결된다.

14.2 Nonproblem: 동일한 디스크가 다른 구조정보를 갖는다 ?

'저는 동일한 10 GB의 IBM 디스크를 갖고 있습니다. 그런데 fdisk는 이들 디스크의 크기상에 차이를 보여줍니다.' 아래처럼 :

```
# fdisk /dev/hdb
Disk /dev/hdb: 255 heads, 63 sectors, 1232 cylinders
Units = cylinders of 16065 * 512 bytes

   Device Boot  Start      End   Blocks   Id  System
/dev/hdb1           1     1232  9896008+  83  Linux native
# fdisk /dev/hdd
Disk /dev/hdd: 16 heads, 63 sectors, 19650 cylinders
Units = cylinders of 1008 * 512 bytes

   Device Boot  Start      End   Blocks   Id  System
/dev/hdd1           1    19650  9903568+  83  Linux native
```

어떻게 이런 결과가 ?

무슨 문제가 생긴 걸까요 ? 무엇보다도 모든 이러한 드라이브는 실제로 10 기가 바이트입니다. hdb는 255*63*1232*512 = 10133544960크기를 갖으며, hdd는 16*63*19650*512 = 10141286400크기를 갖습니다. 그러므로 잘못된 것은 없습니다. 그리고 커널은 이 둘 모두를 10.1 GB로 인식합니다. 그럼 왜 크기상에 차이가 있는 건가요 ? 그것은 커널이 처음 두개의 IDE 디스크의 정보를 BIOS로 부터 가져 오기 때문입니다. 그리고 BIOS는 hdb 를 255 개의 헤드를 갖는 것으로 재할당했기 때문입니다. (and 16*19650/255=1232 cylinders). 여기에서 자리 내림은 약 8 MB 공간을 깍아 먹습니다.

만약 hdd도 동일한 방법으로 재할당되길 원한다면 부팅파라메터를 'hdd=1232,255,63'으로 입력해 주면 됩니다.

14.3 Nonproblem: fdisk 프로그램은 df 프로그램에서 나타내는 것 보다 큰 공간을 표시합니다. ?

fdisk는 디스크상에 얼마나 많은 블록이 있는지를 보여줄 것입니다. 만약 여러분이 디스크상에 파일시스템을 생성시 mke2fs 를 이용하면, 이 파일 시스템은 시스템 용도(bookkeeping)를 위해 약간의 공간을 필요로 합니다. 일반적으로 파일시스템 크기의 4% 정도를 사용합니다. 게다가 mke2fs 실행시 많은 inode를 여러분이 요구하면 더욱더 많이 여분의 공간으로 사용됩니다.

예를 들어:

```
# sfdisk -s /dev/hda9
4095976
# mke2fs -i 1024 /dev/hda9
mke2fs 1.12, 9-Jul-98 for EXT2 FS 0.5b, 95/08/09
...
204798 blocks (5.00%) reserved for the super user
...
# mount /dev/hda9 /somewhere
# df /somewhere
Filesystem         1024-blocks  Used Available Capacity Mounted on
/dev/hda9            3574475      13  3369664      0%   /mnt
# df -i /somewhere
Filesystem           Inodes   IUsed   IFree  %IUsed Mounted on
/dev/hda9            4096000      11 4095989     0%  /mnt
#
```

우리는 4095976개의 블럭을 갖는 파티션을 갖는다. 그리고 이 파티션에 ext2 파일 시스템을 생성한다. 그리고 마우트를 하고나서 이 시스템이 단지 3574475 블럭을 갖는 것을 알게 되었다. 521501 블럭(12%)이 inode와 시스템 용도(bookkeeping) 용도로 이용되었다. 전체크기 3574475 와 사용자가 사용할 수 있는 크기의 차이는 사용중인 13개의 블럭에 루트를 위해 예약된 204798 블럭을 합한것과 같다는 것을 주목하기 바란다. 204798 의 블럭 수치는 tune2fs에 의해 변경 가능하다. 이 '-i 1024'는 단지 news 스풀이나 기타 매우 작은 파일들이 많은 경우에 적당하다. 기본값은 다음과 같다.

```
# mke2fs /dev/hda9
# mount /dev/hda9 /somewhere
# df /somewhere
Filesystem         1024-blocks  Used Available Capacity Mounted on
/dev/hda9            3958475      13  3753664      0%   /mnt
# df -i /somewhere
Filesystem           Inodes   IUsed   IFree  %IUsed Mounted on
/dev/hda9            1024000      11 1023989     0%  /mnt
#
```

이제 단지 137501 blocks (3.3%) 이 inode로 사용된다. 그러므로 우리는 이전보다 384 MB 를 더 사용할 수 있다. (정확하게 각각의 inode는 128 byte를 갖는다) 반면에 이 파일시스템은 이전의 4096000 에 비해 충분한 크기인 1024000 개의 파일을 갖을 수 있다.