8장. 디바이스 드라이버 (Device Drivers)

운영체제의 목적중 하나는 시스템의 하드웨어 장치별로 다른 특징을 사용자로부터 감추는 것이다. 예를 들어 가상 파일 시스템(Virtual File System)은 파일 시스템이 어떤 물리적 장치 에 들었있든 상관없이, 마운트된 파일 시스템들을 일관된 모습으로 보여준다. 이 장에서는 리눅스 커널이 시스템에 있는 물리적인 장치를 어떻게 관리하는지 살펴보기로 한다.

CPU가 시스템에 있는 지능을 가진 유일한 장치는 아니다. CPU 말고도 모든 물리적 장치들 은 지능이 있는 자신만의 하드웨어 컨트롤러를 가지고 있다. 키보드, 마우스, 직렬포트는 SuperIO 칩이 제어하고, IDE 하드디스크는 IDE 컨트롤러가, SCSI 디스크는 SCSI 컨트롤러가 제어한다. 모든 하드웨어 컨트롤러는 각자의 고유한 제어/상태 레지스터(Control and Status Registers, CSRs)를 가지며, 이것은 장치들마다 다르다. Adaptec 2940 SCSI 컨트롤러의 CSRs과 NCR 810 SCSI 컨트롤러의 CSRs는 완전히 다른다. CSRs는 장치를 시작하고 멈추고, 초기화 하며 문제가 발생했을 때 이를 진단하는데 이용된다. 모든 응용프로그램에 하드웨어를 관리 하는 코드를 넣지 않으며, 리눅스 커널만 그 코드를 가지고 있다. 하드웨어 컨트롤러를 다루 고 관리하는 소프트웨어를 디바이스 드라이버라고 한다. 리눅스 커널 디바이스 드라이버는 근본적으로 특권층에서 실행되고, 메모리에 상주하며, 저급 하드웨어 처리 루틴을 가진 공유 라이브러리이다. 리눅스에 있는 디바이스 드라이버는 자신이 관리하는 장치들의 특성들을 처리한다.

유닉스의 기본적인 특징 중의 하나는 장치를 다루는 것을 추상화한다는 것이다. 모든 하드 웨어 장치들은 보통 파일처럼 보이며, 파일을 다루는 데 쓰이는 표준 시스템 콜과 똑같은 함수를 이용하여 열고, 닫고, 읽고, 쓸 수 있다[1]. 시스템의 모든 장치는 장치 특수 파일 (device special file)로 표시가 된다. 예를 들어, 시스템에 있는 첫번째 IDE 디스크는 /dev/hda로 나타낸다. 블럭 (디스크) 장치(block device)나 문자 장치(character device)를 나타 내는 장치 특수 파일은 mknod 명령으로 만들어지며, 메이저와 마이너 장치 번호로 장치를 나타낸다. 네트웍 장치들도 장치 특수 파일로 표시가 되지만, 이들은 리눅스가 시스템에서 네트웍 컨트롤러를 찾아서 초기화할 때 (리눅스에 의해) 만들어진다[2]. 똑같은 디바이스 드라 이버로 제어되는 모든 장치는 똑같은 메이저 장치 번호를 갖는다. 마이너 장치 번호는 다른 장치나 컨트롤러를 구분하는데 사용한다[3]. 예를 들어 첫번째 IDE 디스크의 각 파티션들은 다른 마이너 장치 번호를 갖는다. 그래서 첫번째 IDE 디스크 두번째 파티션은 (/dev/hda2) 메이저 번호로 3, 마이너 번호로 2를 갖는다. 블럭 장치에 있는 파일 시스템을 마운트하는 경우처럼 시스템 콜에 장치 특수 파일을 전달하면, 리눅스는 메이저 장치 번호와 여러 시스 템 테이블을 이용하여(이런 것 중의 하나로 문자 장치 테이블인 chrdevs가 있다), 장치 특 수 파일을 장치의 디바이스 드라이버로 연결한다.

리눅스는 문자, 블럭, 네트웍, 이 세가지 종류의 하드웨어 장치를 지원한다. 문자 장치는 버 퍼를 통하지 하고 바로 읽고 쓸 수 있는 장치로, /dev/cua0과 /dev/cua1 같은 직렬 포 트가 여기에 속한다. 블럭 장치는 일정한 블럭 크기(보통 512 또는 1024 바이트이다)의 배수 로만 읽고 쓸 수 있다. 블럭 장치는 버퍼 캐시(buffer cache)를 통해서 읽고 쓰며, 아무 곳이 나 접근할 수 있다. 즉 어떤 블럭이든 그것이

장치의 어디에 있든지 간에 읽고 쓸 수 있다 는 것이다. 블럭 장치는 장치 특수 파일을 통해서 접근할 수도 있지만, 보통은 파일 시스템 을 통해서 접근한다. 블럭 장치만이 마운트되는 파일 시스템을 지원할 수 있다. 네트웍 장치 는 BSD 소켓 인터페이스로 접근하며, 이는 10장에 있는 네트워킹 서브시스템 부분에서 자 세히 이야기한다.

리눅스 커널에는 많은 서로 다른 디바이스 드라이버가 있지만 (이것이 리눅스의 힘 중의 하 나이다), 그들은 모두 어떤 공통적인 특성을 가지고 있다 :

- **커널 코드** 디바이스 드라이버는 커널의 한 부분이므로, 커널의 다른 코드와 마찬가지로 잘 못되면 시스템에 치명적인 피해를 줄 수 있다. 잘못 만든 드라이버는 시스템을 파괴할 수 있으며, 파일 시스템을 망가트리거나 데이터를 날릴 수도 있다.
- **커널 인터페이스** 디바이스 드라이버는 리눅스 커널이나 자신이 속한 서브시스템에 표준 인 터페이스를 제공해야 한다. 예를 들어, 터미널 드라이버는 리눅스 커널에 파일 I/O 인터 페이스를 제공해야 하며, SCSI 디바이스 드라이버는 커널에 파일 I/O와 버퍼 캐시 인터 페이스를 제공하는 SCSI 서브시스템에 SCSI 장치 인터페이스를 제공해야 한다.
- **커널 메커니즘과 서비스** 디바이스 드라이버는 메모리 할당, 인터럽트 전달, 대기큐같은 표준 커널 서비스를 사용할 수 있다.
- **로더블(Loadable)** 대부분의 리눅스 디바이스 드라이버는 커널 모듈로서, 필요할 때 로드하고 더 이상 필요하지 않을 때 언로드할 수 있다. 이는 커널을 매우 융통성 있게 만들고 시스 템의 자원을 효율적으로 이용할 수 있게 한다[4].
- **설정가능(Configurable)** 리눅스 디바이스 드라이버를 커널에 포함하여 컴파일 할 수 있다. 어떤 장치를 넣을 것인지는 커널을 컴파일할 때 설정할 수 있다[5].
- **동적(Dynamic)** 시스템이 부팅하고 디바이스 드라이버가 초기화 될 때, 시스템은 자신이 제 어할 수 있는 하드웨어 장치를 찾는다. 만약 어떤 디바이스 드라이버가 제어할 수 있는 장치가 없다고 하더라도 문제가 안된다. 이 경우 디바이스 드라이버는 단지 여분으로 있 는 것이고, 시스템 메모리를 조금 잡아 먹는다는 것 말고는 아무런 해도 끼치지 않는다.

8.1 폴링(Polling)과 인터럽트(Interrupt)

장치에 명령을 할 때 (예를 들어 "헤드를 옮겨 플로피 디스크의 42번 섹터를 읽어라"), 디바 이스 드라이버는 그 명령이 언제 끝났는지 아는 방법을 선택할 수 있다. 디바이스 드라이버 는 장치를 폴링할 수도 인터럽트를 사용할 수도 있다.

장치를 폴링한다는 것은 일반적으로 요청한 작업이 끝났는 지를 알기 위해 장치의 상태가 변할 때까지 장치의 상태 레지스터를 계속해서 자주 읽는 것을 말한다. 디바이스 드라이버 는 커널의 한 부분이기 때문에, 만약 드라이버가 폴링만 하려고 한다면 장치가 작업을 끝마 칠 때까지 커널의 다른 부분이 수행될 수 없으므로 끔찍한 일이 벌어질 것이다. 그래서 폴 링을 하는 디바이스 드라이버는 시스템 타이머를 이용하여 어느정도 시간이 지나면 커널이 디바이스 드라이버에 있는 한 루틴을 부르도록 한다. 그러면 이 타이머 루틴은 명령이 수행 되었는지 상태를 검사한다[6]. 이는 리눅스의 플로피 드라이버에서 사용하는 방법이다. 타이머 를 이용하는 폴링은 좋은 방법이지만, 이보다 더 효과적인 방법으로 인터럽트를 사용하는 것이 있다.

제어하는 하드웨어 장치가 서비스를 받아야 할 때 하드웨어 인터럽트를 발생하는 것이 인터 럽트를 이용한 디바이스 드라이버이다. 예를 들어, 이더넷 디바이스 드라이버는 네트웍에서 이더넷 패킷을 받을 때마다 인터럽트를 발생한다. 리눅스 커널은 이 인터럽트를 하드웨어 장치에서 올바른 디바이스 드라이버로 전달할 수 있어야 한다. 이는 디바이스 드라이버가 커널에 인터럽트를 사용하겠다고 등록함으로써 이루어진다. 드라이버는 인터럽트 처리 루틴 의 주소와 자신이 사용하고 싶은 인터럽트 번호를 커널에 등록한다. 현재 디바이스 드라이 버가 어떤 인터럽트를 사용하고 있으며, 그 인터럽트가 얼마나 많이 발생했는지 알려면, /proc/interrupts 파일을 보면 된다.

```
0:      727432          timer
1:      20534           keyboard
2:      0               cascade
3:      79691   +       serial
4:      28258   +       serial
5:      1               sound blaster
11:     20868   +       aic7xxx
13:     1               math error
14:     247     +       ide0
15:     170     +       ide1
```

인터럽트 자원을 요청하는 것은 드라이버가 초기화 될 때 한다[7]. 시스템의 어떤 인터럽트들 은 처음부터 고정되어 있는데, 이는 IBM PC 구조의 오랜 유물이다. 그래서 플로피 컨트롤러 는 언제나 인터럽트 6을 사용한다. 다른 인터럽트들, 예를 들어 PCI 장치에서 발생하는 인 터럽트들은 부팅시에 동적으로 할당된다. 이경우 디바이스 드라이버는 인터럽트의 소유권을 요청하기 이전에 자신이 제어할 장치의 인터럽트 번호 (IRQ)를 먼저 알아내야 한다. 리눅스 는 PCI에서 사용하는 인터럽트에 대해, IRQ 번호를 포함하여 시스템에 있는 장치 정보를 알 수 있는 표준 PCI BIOS 콜백을 지원한다.

인터럽트가 CPU에 어떻게 전달되는지는 하드웨어 구조에 따라 다르지만, 대부분 구조에서 는 시스템에서 다른 인터럽트가 발생하는 것을 막는 특별한 모드에서 인터럽트를 전달한다. 그래서 디바이스 드라이버는 인터럽트 처리 루틴 안에서는 되도록 적은일을 하여, 리눅스 커널이 인터럽트 처리에서 빠져나와 인터럽트되기 전에 하던 일로 되돌아갈 수 있도록 해야 한다. 인터럽트를 받았을 때 많은 일을 해야 하는 디바이스 드라이버는, 나중에 불려도 되는 작업을 커널의 하반부 핸들러나 작업큐에 넣어 처리할 수 있다.

8.2 직접 메모리 접근 (Direct Memory Access, DMA)

데이터를 하드웨어에서 하드웨어 장치로 보내거나 받을 때 인터럽트를 사용하는 디바이스 드라이버는 왔다갔다하는 데이터의 양이 작을 때는 잘 동작한다. 1 밀리초 (1/1000 초)에 한 글자씩 전송하는 9600 bps 모뎀을 예로 들어보자. 만약 인터럽트 처리시간 - 하드웨어 장치 에서 인터럽트가 발생하고, 디바이스 드라이버의 인터럽트 처리 루틴이 불리기까지 걸리는 시간 - 이 작다면 (2 밀리초라고 하자), 데이터 전송으로 전체 시스템에 주는 영향은 매우 작을 것이다. 9600 bps 모뎀의 데이터 전송은 겨우 CPU 프로세서 시간의 0.002% 만을 이용 할 뿐이다. 그러나 하드디스크 컨트롤러나 이더넷 장치같이 고속도 장치들의 데이터 전송률 은 매우 높다. SCSI 장치는 1초에 40MB까지 데이터를 전송할 수 있다.

DMA는 이런 문제를 해결하기 위해 개발되었다. DMA 컨트롤러는 CPU가 개입하지 않고 장치와 시스템의 메모리 사이에 데이터를 전송할 수 있도록 한다. PC의 ISA DMA 컨트롤러 는 여덟개의 DMA의 채널을 가지고 있으며, 이 중 7개를 디바이스 드라이버가 사용할 수 있다. 각 DMA 채널은 16 비트 주소 레지스터와 16 비트 카운터 레지스터에 연결되어 있다. 데이터 전송을 초기화하기 위해 디바이스

드라이버는 DMA 채널의 주소레지스터와 카운터 레지스터, 데이터 전송 방향(읽을 것인지, 쓸 것인지)을 함께 설정한다. 그리고 자신이 원할 때 장치에게 DMA를 시작해도 좋다고 명령한다. 데이터 전송이 완료되면 장치는 PC에 인 터럽트를 발생한다. 전송이 이루어지는 동안에 CPU는 다른일을 맘대로 할 수 있다.

디바이스 드라이버는 DMA를 매우 조심해서 사용해야 한다. 무엇보다도 DMA 컨트롤러는 가상 메모리에 대해서 아무것도 모르고 있으며, 그저 시스템의 물리적 메모리에 접근할 뿐 이다. 따라서 DMA에서 사용하는 메모리는 물리적인 메모리에서 연속된 블럭으로 되어 있 어야 한다. 이는 프로세스의 가상 메모리 주소공간으로 DMA를 바로 사용할 수 없다는 말 이다[8]. 어쨌든 사용자는 프로세스의 물리적 페이지를 메모리에 락(lock)을 걸어[9], DMA 작업 중에 메모리가 스왑 장치로 스왑 아웃되는 것을 방지하게 만들수 있다. 둘째로, DMA 컨트 롤러는 물리적 메모리 전체에 접근할 수 없다. DMA 채널의 주소 레지스터는 DMA 어드레 스의 처음 16 bit를 나타내고, 페이지 레지스터에 다음 8 비트가 있다. 즉 DMA 가 사용할 수 있는 메모리는 하부 16MB로 제한되어 있다는 것이다.

DMA 채널은 오직 7개 밖에 사용할 수 없고, 디바이스 드라이버들이 같이 공유할 수 없는 드문 자원이다. 인터럽트와 마찬가지로 디바이스 드라이버는 어떤 DMA 채널을 사용할 지 를 알아야 한다. 역시 인터럽트에서처럼 어떤 장치가 사용하는 DMA 채널은 고정되어 있다. 예를 들어, 플로피 장치는 항상 DMA 채널 2번을 사용한다. 가끔은 장치가 사용하는 DMA 채널은 점퍼로 설정할 수 있다. 많은 이더넷 장치들은 이런 기술을 사용한다. 이보다 더 융 통성 있는 장치들은 어떤 DMA를 채널을 사용할 것인지 알려줄 수 있어서 (자신의 CSRs을 통하여), 디바이스 드라이버는 단지 비어있는 DMA 채널을 사용하면 된다.

리눅스는 DMA 채널 하나당 있는 dma_chan 자료구조의 벡터를 이용하여 DMA 채널의 사 용여부를 추적할 수 있다. dma_chan 자료구조는 두개의 항목으로 되어 있는데, 하나는 DMA 채널의 소유자를 나타내는 문자열이고, 다른 하나는 DMA 채널이 할당되어 있는지 비 어 있는지를 나타내는 플래그이다. cat /proc/dma라는 명령을 내리면 나오는 것이 이 dma_chan 자료구조의 벡터이다.

8.3 메모리

디바이스 드라이버는 메모리를 사용할 때 주의해야 한다. 디바이스 드라이버는 리눅스 커널 의 일부분이므로 가상 메모리를 사용할 수 없다. 디바이스 드라이버가 실행될 때, 즉 인터럽 트를 받았다던지 하반부 핸들러(bottom half handler)나 작업큐 핸들러(task queue handler)가 스 케쥴되었을 때, current 프로세스는 바뀔 수 있다[10]. 디바이스 드라이버는 특정한 프로세스 가 실행되고 있을 때, 비록 그 프로세스의 한켠에서 돌아가고 있더라도, 그 특정 프로세스에 의존할 수 없다. 커널의 나머지 부분처럼 디바이스 드라이버도 자료구조를 만들어 자신이 제어하는 장치를 관리해야 한다. 이러한 자료구조는 정적으로 할당하여 디바이스 드라이버 의 코드의 일부로 포함될 수도 있지만, 이는 커널을 필요이상으로 크게 만들어 낭비적이다. 대부분의 디바이스 드라이버는 커널의 페이지되지 않는 메모리(non-paged)를 할당받아 자신 의 자료를 넣는다.

리눅스는 커널 메모리를 할당하고 해제하는 루틴을 제공하는데, 디바이스 드라이버는 이를 사용한다. 커널 메모리는 2의 제곱승 단위로 할당된다. 예를 들면 128이나 512 크기로 할당 되는데, 디바이스 드라이버가 더 작은 크기를 요청해도 이렇게 할당된다. 디바이스 드라이버 가 요청하는 크기는 다음 블럭의 크기에 맞춰 올림하여 할당된다. 이렇게 하면 프리 블럭들 을 합쳐 더 큰 블럭을 만들 수 있으므로, 커널 메모리 해제가 쉬워진다[11].

커널 메모리를 요청받았을 때 리눅스는 몇가지 여분의 일을 해야된다. 만약 프리 메모리가 적으면, 물리적 페이지를 폐기하거나 스왑 장치로 스왑 아웃해야 한다. 일반적으로 리눅스는 메모리를 요청한 프로세스를 잠시 보류시키고, 충분한 물리적 메모리가 생길 때까지 작업을 대기큐에 넣어둔다. 어떤 디바이스 드라이버(또는 실제 리눅스 커널 코드)는 이런 작업이 발 생하는 것은 원하지 않으며, 이 경우 곧바로 메모리를 할당할 수 없다면 커널 메모리 할당 루틴은 실패할 수도 있다. 만약 디바이스 드라이버가 할당받은 메모리를 DMA로 입출력을 하기를 원한다면, 그 메모리를 요구할 때 DMA가능이라고 지정할 수 있다. 이렇게 한 시스 템에 DMA가능 메모리를 구성하는 것을 알아야 하는 것은 리눅스 커널이지 디바이스 드라 이버가 아니다.

8.4 커널과 디바이스 드라이버와의 인터페이스

리눅스 커널은 디바이스 드라이버들과 표준적인 방법을 통하여 상호작용할 수 있어야 한다. 모든 종류의 디바이스 드라이버 - 문자, 블럭, 네트웍 디바이스 드라이버 - 는 커널이 이들 로에게 서비스를 요청할 때 사용할 수 있는 공통적인 인터페이스를 제공한다. 이 공통적인 인터페이스는 커널이 서로 많이 다른 장치들과 디바이스 드라이버를 완전히 똑같이 다룰 수 있게 한다. 예를 들어 SCSI와 IDE 디스크는 매우 다르게 동작하지만, 리눅스 커널은 똑같 은 인터페이스를 통해 이들을 사용한다.

리눅스는 매우 동적이다. 리눅스 커널은 부팅할 때마다 다른 물리적 장치들을 알게 되고, 다 른 디바이스 드라이버를 필요로 하게 된다. 리눅스는 커널을 빌드할 때 설정 스크립트를 통 하여 여러 디바이스 드라이버를 포함할 수 있게 한다. 이렇게 들어간 디바이스 드라이버는 부팅할 때 초기화가 되는데, 이들이 제어할 하드웨어가 없을 수도 있다. 어떤 드라이버들은 커널 모듈로 만들어져서 자신이 필요로 할 때에만 로드될 수 있다. 이러한 디바이스 드라이 버의 동적인 성격을 원할하게 하기 위해, 디바이스 드라이버는 자신이 초기화될 때 커널에 자기 자신을 등록한다. 리눅스는 디바이스 드라이버와의 인터페이스의 한 부분으로서, 등록 된 디바이스 드라이버의 테이블을 관리한다. 이들 테이블은 해당하는 종류의 장치와 인터페 이스를 제공하는 함수들의 포인터와 정보를 가지고 있다.

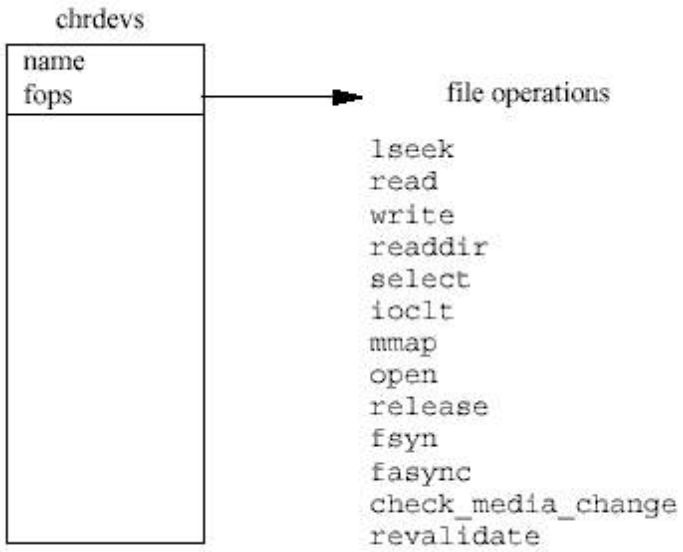

그림 8.1 : 문자 장치

8.4.1 문자 장치(Character Device)

문자 장치는 리눅스의 장치들 중에서 가장 단순한 것이다. 프로그램은 그 장치가 마치 파일 인 것처럼 표준 시스템 콜을 사용하여 열고, 읽고, 쓰고, 닫을 수 있다. 이러한 사실은 그 장 치가 PPP 데몬이 리눅스 시스템을 인터넷에 연결하기 위해 사용하는 모뎀이라 할 지라도 마찬가지다. 문자 장치가 초기화될 때 이것의 디바이스 드라이버는, device_struct 자료 구조의 벡터인 chrdevs에 자신의 엔트리를 추가함으로써 리눅스 커널에 자신을 등록한다[12] . 장치의 메이저 장치 번호는 (예를 들어 tty 장치에 할당되는 4번) 이들 배열의 인덱스 로서 사용된다. 장치에 대한 메이저 장치 번호는 고정되어 있다. chrdevs 벡터의 각 원소인 device_struct 자료구조는 두가지 항목을 가지고 있다. 하나는 디바이스 드라이버의 등 록이름에 대한 포인터이고, 다른 하나는 파일 연산 블럭에 대한 포인터이다. 이 파일 연산 블럭은, 파일을 열고, 쓰고, 읽고, 닫는 이런 파일 연산을 수행하는 문자 디바이스 드라이버 에 있는 루틴의 주소들이다[13]. /proc/devices에 있는 문자 장치에 대한 내용들은 모두 chrdevs 벡터에서 가져온 것이다.

문자 장치 (예를 들어 /dev/cua0)를 나타내는 문자 특수 파일을 열면, 커널은 올바른 문자 디바이스 드라이버의 파일 처리 루틴이 불릴 수 있도록 셋업을 해주어야 한다. 보통의 파일 이나 디렉토리처럼 각 장치 특수 파일은 VFS inode로 표현된다. 문자 특수 파일에 대한 VFS inode는 장치의 메이저 식별자와 마이너 식별자를 가지고 있다 (이는 모든 장치 특수 파일에서 동일하다). 이 VFS inode는 장치 특수 파일을 조회한 경우에, 실제 기반하는 파일 시스템이 (EXT2같은) 파일 시스템에 실제로 있는 정보를 가지고 만든다.

각 VFS inode는 한 셋트의 파일 연산들과 연결되어 있는데, 이들 연산은 그 inode가 가리키 는 파일 시스템 객체에 따라 다르다[14]. 문자 특수 파일을 나타내는 VFS inode가 만들어질 때 마다, 이 inode의 파일 연산 함수들은 기본 문자 장치 연산으로 설정된다. 이는 단 하나의 파일 연산 - 파일 열기 연산만 가지고 있다. 응용프로그램이 문자 특수 파일을 열면, 이 포 괄적인 열기 파일 연산 함수는, 장치의 메이저 식별자를 chrdevs 벡터에 대한 인덱스로 사 용하여, 이 장치에 대한 파일 연산 블럭을 가져온다. 또한 이 문자 특수 파일을 설명하는 file 자료구조의 파일 연산 포인터가 디바이스 드라이버의 것을 가리키도록 이 자료구조를 셋업한다. 이후, 응용프로그램에서 부르는 모든 파일 연산은 문자 장치의 파일 연산으로 매 핑되어 불리게 된다.

8.4.2 블럭 장치(Block Device)

블럭 장치들도 파일처럼 접근하는 것을 지원한다. 열린 블럭 특수 파일에 올바른 파일 연산 세트를 제공하는데 사용되는 방법은 문자 장치에 사용했던 방법과 매우 흡사하다. 리눅스는 blkdevs 벡터로 등록된 블럭 장치들을 관리한다. blkdevs는 chrdevs 벡터에서와 마찬가 지로 장치의 메이저 장치번호로 인덱스되어 있다. 또한 그 엔트리 역시 device_struct 자료구조이다. 문자 장치와 다른 점은, 블럭 장치들의 클래스라는게 있다는 것이다. SCSI 장 치나 IDE 장치 같은 것이 그런 클래스이다. 클래스는 리눅스 커널에 자신을 등록하고 커널 에 파일 함수들을 제공한다. 어떤 클래스의 블럭 장치들에 사용하는 디바이스 드라이버는 클래스 고유의 특별한 클래스 인터페이스를 제공한다. 그래서 예를 들어 SCSI 디바이스 드 라이버는, SCSI 서브시스템이 커널에 해당 장치에 대한 파일 함수를 제공하는데 사용할 수 있는 인터페이스를 SCSI 서브시스템에 제공해야 한다.

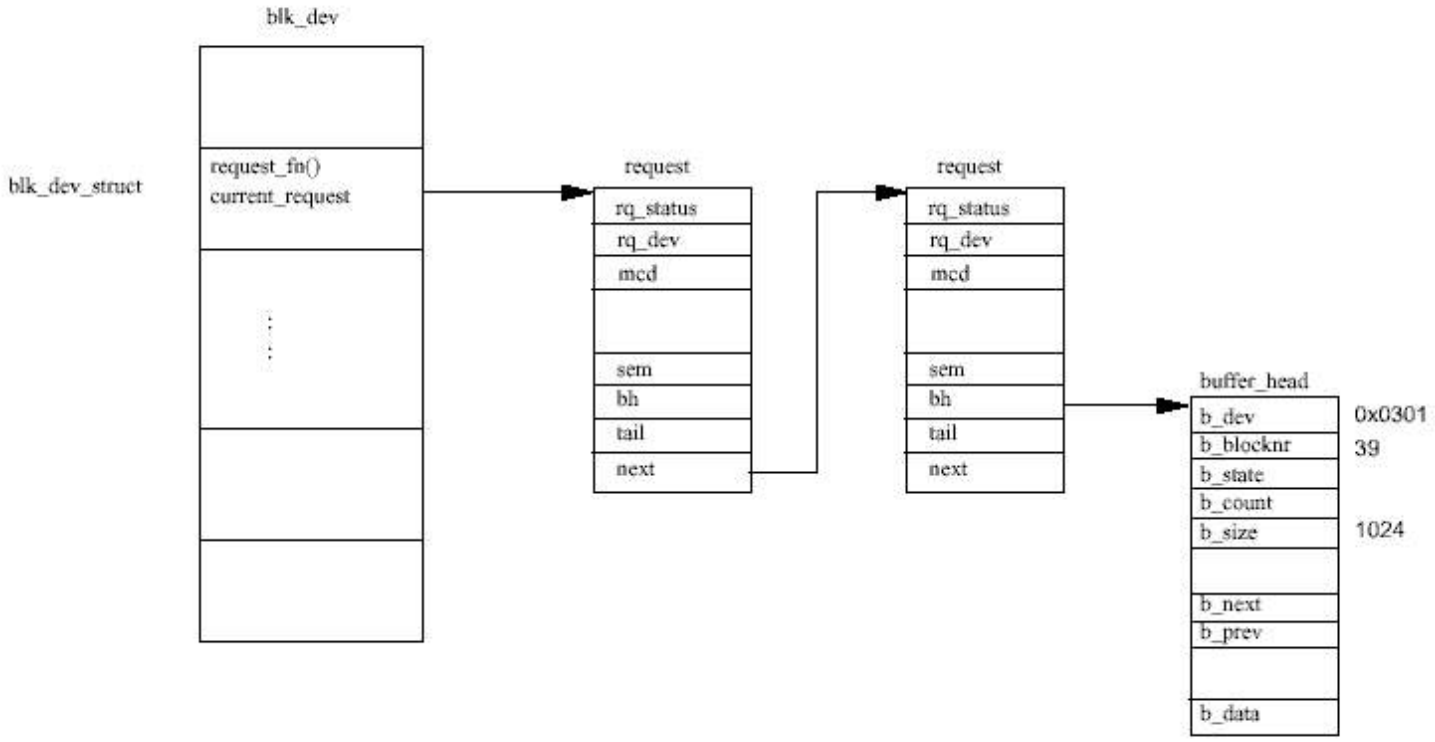

그림 8.2 : 버퍼 캐시 블럭 장치 요청

모든 블럭 디바이스 드라이버는 보통의 파일 연산과 함께 버퍼 캐시에 대한 인터페이스를 제공해야 한다[15]. 각 블럭 디바이스 드라이버는 blk_dev 벡터에 있는 blk_dev_struct 자 료구조의 내용을 채운다. 여기에서도, 이 벡터에 대한 인덱스는 장치의 메이저 번호이다. blk_dev_struct 자료구조는 요청 (request) 루틴의 주소와, 버퍼 캐시가 한 블럭의 데이터 를 읽거나 쓰기 위해 드라이버에게 하는 요청을 나타내는 request 자료구조 리스트에 대 한 포인터로 구성되어 있다.

버퍼 캐시는 등록된 장치에서 데이터를 읽거나 쓰려고 할 때, request 자료구조를 blk_dev_struct 에 추가한다. 그림 8.2는 각request가 하나 이상의 buffer_head 자 료구조에 대한 포인터를 가지고 있고, 각 request는 한 블럭의 데이터를 읽거나 쓰라는 요청이라는 것을 보여준다. buffer_head 자료구조는 버퍼 캐시에 의해 락이 되며, 이 버퍼 로의 블럭 연산이 끝나길 기다리는 프로세스가 있을 것이다. 각 request 구조체는 정적 리 스트인 all_requests 리스트에서 할당된다. 요청이 텅빈 요청 리스트에 추가되면, 이 요청 큐를 처리하기 위해 드라이버의 요청 함수가 불리게 된다. 그러면 드라이버는 그저 단순히 리스트에 있는 모든 request를 처리할 것이다.

일단 디바이스 드라이버가 요청을 처리하고 나면, 드라이버는 request 구조체에서 각각의 buffer_head 구조체를 없애고, 이것이 갱신되었음을 표시하고 이들의 락을 해제해야 한다. 이렇게 buffer_head의 락을 해제하면, 그 블럭 연산이 끝나길 기다리며 잠들어있는 프로 세스가 있을 때 이를 깨우게 된다. 이런 예로 파일 이름을 해결하는 과정에서 EXT2 파일 시스템이 파일 시스템을 담고 있는 블럭 장치로부터 다음 EXT2 디렉토리 엔트리를 포함하 고 있는 데이터 블럭을 읽어야 하는 경우가 있다. 프로세스는 디바이스 드라이버가 자신을 깨울 때까지 잠들게 되며, 깨어났을 때에는 buffer_head에 디렉토리 엔트리가 들어있을 것이다. 이제 request 자료구조는 비었다고 표시되고, 이 자료구조는 이제 다른 블럭 요청 을 위해 사용될 수 있게 된다.

8.5 하드 디스크(Hard Disk)

디스크 드라이브는 자료를 회전하는 디스크 원반(platter)에 저장함으로써 자료를 좀더 영속 적으로 저장할 수 있게 한다. 자료를 기록하기 위해 아주 조그만 헤드가 원반의 표면에 있 는 미세한 알갱이를 자성을 띄게 한다. 헤드는 특정 미세한 알갱이의 자성을 감지하여 자료 를 읽는다.

디스크 드라이브는 하나 이상의 원반(platter)으로 구성되어 있고, 각 원반은 미세하게 간 유 리나 세라믹 복합물질에 미세한 산화철이 얇은 층으로 코팅되어 있다. 원반들은 가운데 축 (spindle)에 연결되어 일정한 속도로 회전을 하는데, 이 회전 속도는 모델에 따라서 3000RPM 부터 10000RPM까지 다르다. 이를 단지 360RPM으로 회전하는 플로피 디스크와 비교해보면 그 차이를 느낄 수 있을 것이다. 디스크의 읽기/쓰기 헤드는 자료를 읽고 쓰는 역할을 하며, 각 표면마다 하나의 헤드가 있어 각 원반에 헤드가 쌍으로 존재한다. 읽기/쓰기 헤드는 물 리적으로 원반의 표면을 건드리지 않으며, 대신 아주 얇은 (백만분의 10인치 정도) 공기 쿠 션 위에 떠있다. 읽기/쓰기 헤드는. 헤드를 움직이는 장치(actuator)에 의해 원반의 표면을 가 로질러 움직인다. 모든 읽기/쓰기 헤드는 서로 붙어 있어서 원반의 표면에서 똑같이 움직이 게 된다.

원반의 각 표면은 트랙(track)이라고 하는 아주 가는 동심원으로 나누어진다. 트랙0은 가장 바깥에 있는 트랙이고, 가장 높은 번호를 갖는 트랙은 중심 축에 가장 가까운 트랙이다. 실 린더(cylinder)는 똑같은 번호를 가지는 트랙의 집합이다. 따라서 디스크에 있는 모든 원반의 양쪽에 있는 5번째 트랙은 모두 5번 실린더이다. 실린더의 개수는 트랙의 개수와 같으므로 종종 디스크의 기하적 구조를 설명할 때 실린더라는 용어를 쓰는 것을 볼 수 있을 것이다. 각 트랙은 섹터(sector)로 나뉜다. 섹터는 자료를 하드디스크에 저장하고 읽어들일 수 있는 최소단위로 디스크의 블럭 크기와 같다. 일반적인 섹터크기는 512바이트이고, 디스크를 제 작한 후 포맷을 할 때 이 크기가 지정된다.

디스크는 보통 기하적 구조 - 실린더와 헤드, 그리고 섹터의 개수 - 로 이야기한다. 예를 들 어 부팅할 때 리눅스에서 필자의 IDE 디스크 중의 하나를 다음과 같이 나타난다.

hdb: Conner Peripherals 540MB-CFS540A, 516MB w/64kB Cache, CHS=1050/16/63

이것은 디스크가 1050개의 실린더 (트랙), 16개의 헤드 (8개의 원반), 그리고 트랙마다 63 개의 섹터를 가지고 있다는 것을 말한다. 각 섹터 즉 블럭마다 512바이트의 크기를 가지므 로, 디스크의 저장용량은 529200 바이트가 된다. 이는 위에서 보여주는 디스크의 용량 516MB하고는 차이가 있는데, 이는 섹터 중의 일부는 디스크 파티션 정보를 간직하는데 사 용되기 때문이다. 어떤 디스크들은 자동으로 배드 섹터(bad sector)를 찾아내서 디스크가 제대 로 작동하도록 인덱스를 다시 붙이기도 한다.

하드 디스크는 파티션(partition)으로 쪼개질 수 있다. 파티션은 특별한 목적을 위해 할당한 섹터들의 거대한 그룹이다. 디스크의 파티션을 나누는 것은 디스크를 여러 운영체제로 쓰거 나, 다른 목적으로 사용할 수 있도록 해준다. 많은 리눅스 시스템은 하나의 디스크에 세개의 파티션을 가진다 - 하나는 DOS 파일 시스템이고, 다른 하나는 EXT2 파일 시스템을, 마지막 은 스왑 파티션이다. 하드디스크의 파티션 정보는 파티션 테이블에 적혀있다. 파티션 테이블 의 각 엔트리는 파티션이 어디서 시작하고 어디서 끝나는지를 헤드와 섹터, 실린더 번호를 가지고 기술한다. fdisk로 DOS로 포맷된 디스크는 4개의 1차 디스크 파티션(primary disk partition)을 가질 수 있다. 파티션 테이블의 4개 엔트리 모두가 쓰여야 하는 것은 아니다. fdisk는 세가지 유형의 파티션을 지원하는데, 각각 1차(primary), 확장(extended), 논리 (logical) 파티션이다. 확장 파티션은 진짜 파티션이 아니라, 여러 개의 논리 파티션을 가지고 있는 것이다. 확장파티션과 논리 파티션은 1차 파티션을 네개밖에 가질 수 있는 제한을 우 회하기 위한 방법으로 개발되었다. 다음은 두개의 1차 파티션을 가지고 있는 디스크에 대해 fdisk를 실행했을 때의 출력 결과이다 :

```
Disk /dev/sda:  64 heads,  32 sectors, 510 cylinders
Units = cylinders of 2048 * 512 bytes

        Device  Boot    Begin   Start   End     Blocks  Id      System
        /dev/sda1               1       1       478     489456  83      Linux native
        /dev/sda2               479     479     510     32768   82      Linux swap

Expert command (m for help) : p

Disk /dev/sda:  64 heads,  32 sectors, 510 cylinders

        Nr      AF      Hd      Sec     Cyl     Hd      Sec     Cyl     Start   Size    ID
        1       00      1       1       0       63      32      477     32      978912  83
        2       00      0       1       478     63      32      509     978944  65536   82
        3       00      0       0       0       0       0       0       0       0       00
        4       00      0       0       0       0       0       0       0       0       00
```

이는 첫번째 파티션이 실린더 0 (즉 트랙 0), 헤드 1, 섹터 1에서 시작하며, 477번 실린더, 32 개의 섹터, 63개의 헤드까지 있다는 것을 보여준다. 여기엔 트랙당 32개의 섹터와 64개의 읽기/쓰기 헤드가 있으므로, 이 파티션의 크기는 있어서 실린더 개수와 같다[16]. fdisk는 기본 적으로 파티션을 실린더를 경계로 배열한다. 이는 맨 바깥 실린더 0에서 시작하여 축이 있 는 방향으로 안쪽으로 들어가 478개의 실린더를 갖는다. 두번째 파티션은 스왑 파티션으로 서 다음 실린더 (478)에서 시작하여 디스크의 가장 안쪽 실린더까지 뻗쳐있다.

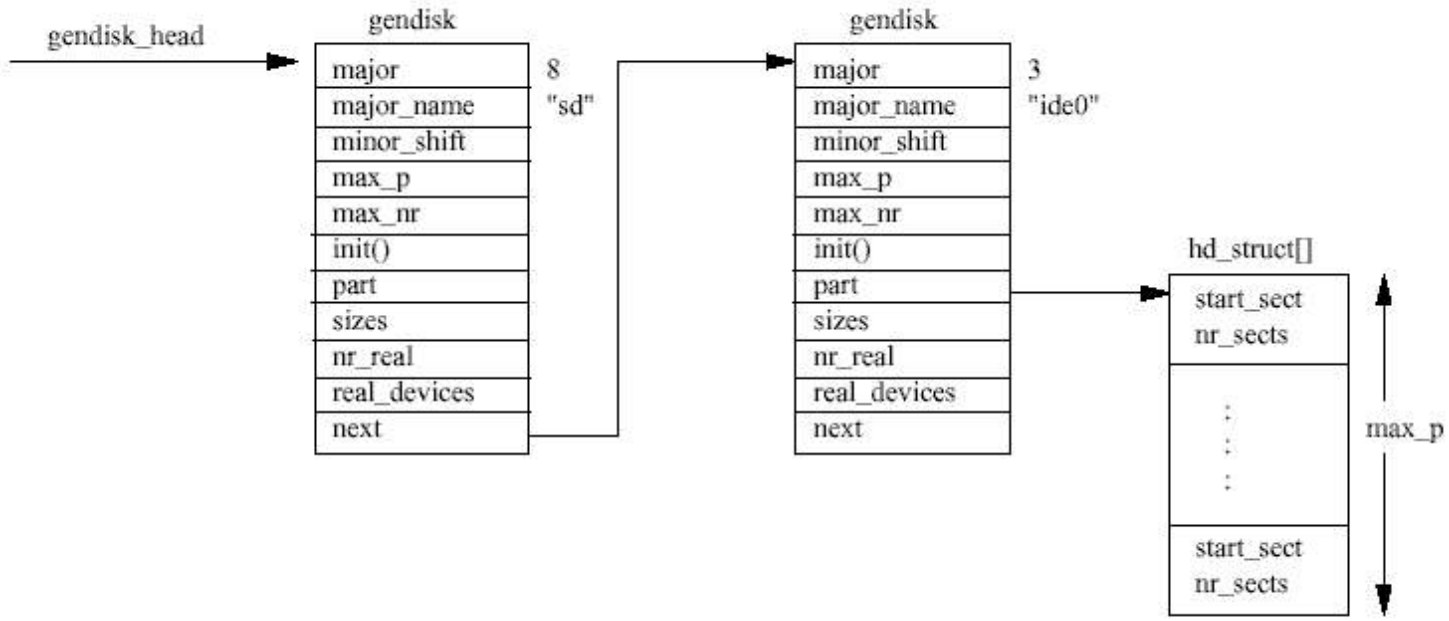

그림 8.3 : 디스크의 연결 리스트

리눅스는 초기화하는 동안 시스템에 있는 하드디스크의 배치도를 그려 이를 매핑한다. 먼저 시스템에 하드디스크가 몇 개가 있고 어떤 종류인지 알아낸다. 나아가 개별 디스크의 파티 션이 어떻게 나누어 졌는지도 찾아낸다. 이들은 gendisk 자료 구조로 표시되며, 이들의 리 스트는 gendisk_head 리스트 포인터가 가리키고 있다. IDE같은 개별 디스크 서브시스템 은 초기화될 때 자신이 찾은 디스크를 gendisk 자료구조로 만들어낸다. 디스크 서브시스템 은 이를 파일연산을 등록하고 엔트리를 blk_dev 자료구조에 넣을 때와 동시에 한다. 각 gendisk 자료구조는 고유한 메이저 장치 번호를 가지며, 이는 블럭 특수 장치의 메이저 번 호와 일치한다. 예를 들어, SCSI 디스크 서브시스템은 모든 SCSI 디스크 장치에 적용되는 메 이저번호 8을 가지는 하나의 gendisk 엔트리 ("sd")를 만든다. 그림 8.3은 두개의

gendisk 엔트리를 보여준다. 앞의 것은 SCSI 디스크 서브시스템의 것이고, 다음 것은 IDE 디스크 컨트롤러 것이다. 이것은 첫번째 IDE 컨트롤러인 ide0이다.

디스크 서브시스템이 초기화할 때 만드는 gendisk 엔트리는, 단지 리눅스가 파티션을 검사 할 때에만 쓰인다. 대신, 각 디스크 서브시스템은 장치의 메이저와 마이너 장치 번호를 물리 적인 디스크에 있는 파티션과 매핑시킬 수 있도록 자신만의 자료구조를 구축한다. 블럭 장 치가 버퍼 캐시나 파일 연산을 통해 읽혀지거나 쓰일 때, 커널은 이 연산을 블럭 장치 특수 파일(예를 들어 /dev/sda2)에서 발견한 메이저 장치번호를 이용하여 올바른 장치로 보내 게 된다. 마이너 장치 번호를 실제 물리적 장치에 연결하는 것은 개별 디바이스 드라이버나 서브시스템의 역할이다.

8.5.1 IDE 디스크

지금 리눅스 시스템에서 가장 일반적으로 사용하는 디스크는 IDE(Integrated Disk Electronics) 디스크이다. IDE는 SCSI같은 I/O 버스가 아니라 디스크 인터페이스이다[17]. 각 IDE 컨트롤러 는 두개까지 디스크를 지원할 수 있다. 하나는 주(master) 디스크이고 다른 하나는 종속 (slave) 디스크이다. 주이냐 아니면 종속이냐는 보통 디스크에 있는 점퍼로 설정한다. 시스템 에 있는 첫번째 IDE 컨트롤러는 1차 (primary) IDE 컨트롤러라고 하고 다음 것은 2차 (secondary) 컨트롤러라고 한다. IDE는 1초에 3.3 Mbyte의 데이터를 전송할 수 있으며, IDE 디스크의 최대 크기는 538 MB이다. 확장 IDE (Extended IDE, EIDE)는 디스크의 크기를 최대 8.6 GB, 전송속도를 초당 16.6 MB까지 올린 것이다. IDE와 EIDE 디스크는 SCSI 디스크보다 싸서 근래의 대부분의 PC는 보드상에 하나 이상의 IDE 컨트롤러를 가지고 있다.

리눅스는 IDE 디스크의 이름을 디스크 컨트롤러를 발견한 순서에 따라 붙인다. 1차 컨트롤 러의 주 디스크는 /dev/hda, 종속 디스크는 /dev/hdb이다. /dev/hdc는 2차 IDE 컨트 롤러에 있는 주 디스크이다. IDE 서브시스템은 리눅스 커널에 IDE 컨트롤러를 등록하지만 디스크를 등록하는 것은 않는다. 1차 IDE 컨트롤러에 대한 메이저 식별자(또는 장치 번호) 는 3이고, 2차 IDE 컨트롤러는 22이다. 그래서 시스템이 두개의 IDE 컨트롤러를 가지고 있 다면, blk_dev와 blkdevs 벡터의 3번과 22번 인덱스에 IDE 서브시스템의 엔트리가 있을 것이다. IDE 디스크의 블럭 특수 파일은 이런 번호 붙이는 방법을 반영하여, 1차 IDE 컨트 롤러에 연결되어 있는 /dev/hda와 /dev/hdb 두 개는 메이저 식별자로 3을 가진다. 커널은 이들 블럭 특수 파일을 관리하는 IDE 서브시스템에 대한 파일 연산이나 버퍼 캐시 연산 을, 메이저 식별자를 인덱스로 사용하여 알아낸 IDE 서브시스템으로 전달한다. 어떤 요청이 들어왔을 때 어떤 IDE 디스크에게 요청이 들어왔는지 알아내는 것은 IDE 서브시스템의 몫 이다. 이를 위해 IDE 서브시스템은 장치 특수 식별자에 있는 마이너 장치 번호를 사용하는 데, 이것은 올바른 디스크의 해당하는 파티션으로 요청을 보낼 수 있도록 하는 정보를 가지 고 있다. 1차 IDE 컨트롤러에 있는 종속 IDE 드라이브인 /dev/hdb의 장치 식별자는 (3,64)이고, 이 디스크의 첫번째 파티션(/dev/hdb1)에 대한 장치 식별자는 (3,65)이다.

8.5.2 IDE 서브시스템의 초기화

IDE 디스크는 IBM PC의 역사의 많은 부분을 함께 해왔다. 이 시간을 통해 이들 장치로의 인터페이스도 변해 왔으며, 이는 IDE 서브시스템의 초기화를 처음 생각했던 것보다 더 복잡 하게 만든다.

리눅스가 지원할 수 있는 최대 IDE 컨트롤러의 갯수는 4개이다. 각 컨트롤러는 ide_hwifs 벡터에 있는 ide_hwif_t 자료구조로 표현한다. 각 ide_hwif_t 자료구조는 두개의 ide_drive_t 자료 구조를 가지고 있으며, 이 중 하나는 주 IDE 드라이브, 다른 하 나는 종속 IDE 드라이브를 위한 것이다. IDE 서브시스템를 초기화할 때 리눅스는 먼저 시스 템의 CMOS 메모리에 디스크 정보가 있는지 살펴본다. CMOS 메모리는 밧데리에서 전원을 공급받기 때문에 PC의 전원을 끄더라도 내용물을 잃어버리지 않는 메모리다. 이 CMOS 메 모리는, PC의 전원이 켜져 있는지 꺼져 있는지에 무관하게 돌아가는 실시간 시계(real time clock, RTC) 장치에 들어있는 것이다. CMOS 메모리의 위치는 시스템의 BIOS에서 설정하며, 이는 어떤 IDE 컨트롤러와 드라이브가 있는지 리눅스에게 알려준다. 리눅스는 발견한 디스 크의 기하적 정보를 BIOS로부터 얻으며, 이 정보를 이 드라이브에 대한 ide_hwif_t 자료 구조를 설정하는데 사용한다. 최근에 나온 PC들은 PCI EIDE 컨트롤러를 포함하고 있는 Intel 82430 VX 칩셋같은 PCI 칩셋을 사용한다. 이 경우 IDE 서브시스템은 PCI BIOS 콜백을 이용하여 시스템에 있는 PCI (E)IDE 컨트롤러를 찾는다. 그리고 현재 있는 이들 칩셋에 PCI 고유의 질의 루틴을 호출한다.

IDE 인터페이스, 즉 컨트롤러가 발견되면, 컨트롤러와 이에 연결된 디스크를 반영하여 ide_hwif_t가 설정된다. IDE 드라이버가 I/O 메모리 공간에 있는 IDE 명령 레지스터에 명 령을 씀으로써 동작이 이루어진다. 1차IDE 컨트롤러의 명령 레지스터와 제어 레지스터의 기본 I/O 주소는 0x1F0 - 0x1F7이다. 이들 주소는 IBM PC 초창기에서부터 관행으로 설정된 것이다. IDE 드라이버는 각 컨트롤러를 리눅스 블럭 버퍼 캐시와 VFS에 등록하는데, 이는 blk_dev와 blkdevs 벡터에 추가하는 것이다. IDE 드라이버는 또한 해당하는 인터럽트에 대한 제어권을 요청한다. 이들 인터럽트 역시 관행처럼 1차 IDE 컨트롤러에 14, 2차 IDE 컨트롤러는 15로 설정된다. 그렇지만, 이들 설정은 IDE의 다른 상세한 설정과 마찬가지로 커널에 명령행(command line) 옵션을 주어서 덮어 쓸 수 있다. IDE 드라이버는 또한 부팅시 발견된 IDE 컨트롤러마다 gendisk를 만들어 gendisk의 리스트에 추가한다 이 리스트는 나중에 부팅시 발견된 모든 하드 디스트의 파티션 테이블을 찾는데 사용한다. 파티션을 검 사하는 코드는 IDE 컨트롤러가 두개의 IDE 디스트를 제어할 수도 있다는 것을 알고 있다.

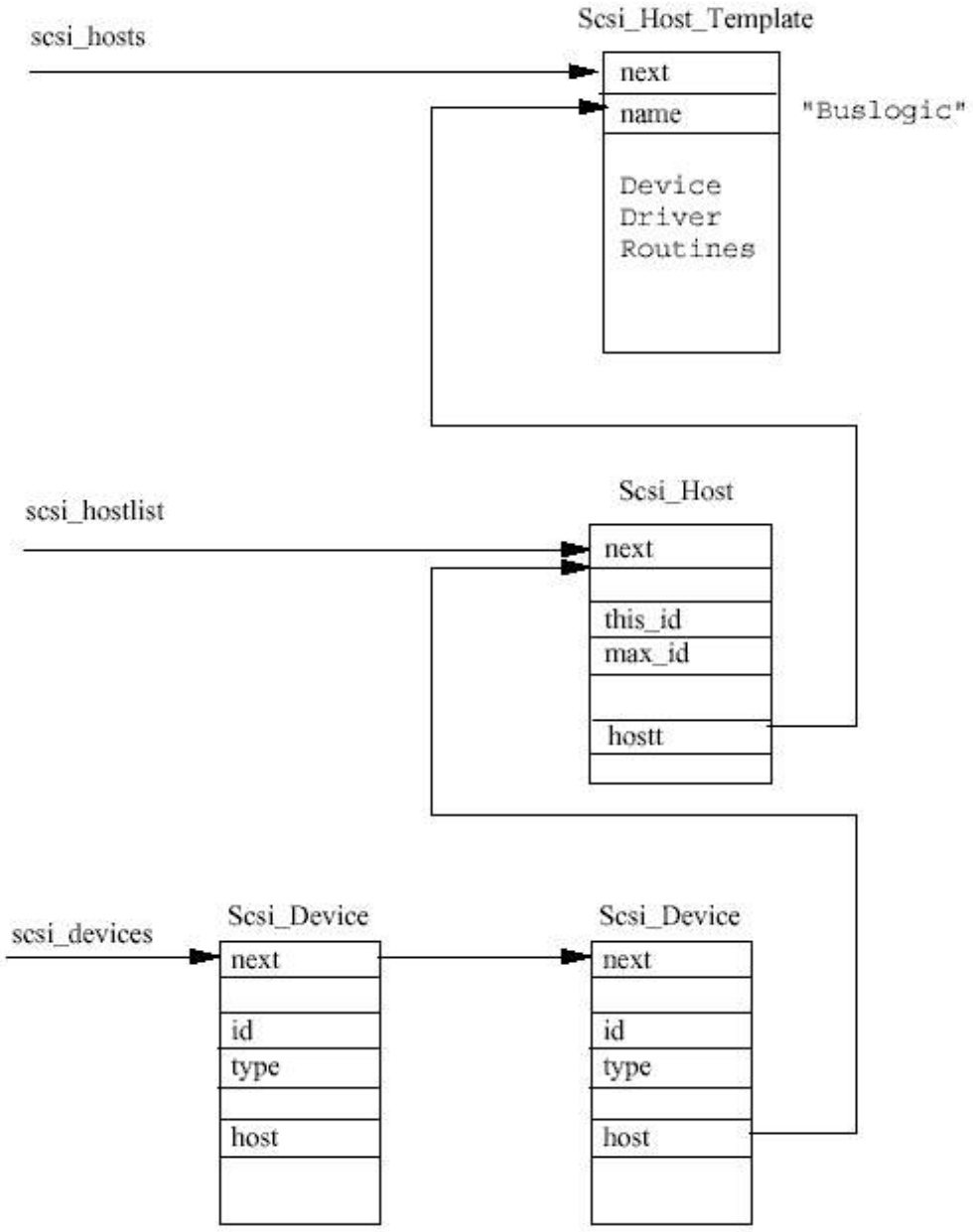

그림 8.4 : SCSI 자료구조

8.5.3 SCSI 디스크

SCSI (Small Computer System Interface) 버스는, 하나 이상의 호스트를 포함하여 버스마다 8개 까지의 장치를 지원할 수 있는 효율적인 1대1 데이터 버스이다. 각 장치는 고유한 식별자 를 가져야 하는데, 이는 대개 디스크에 있는 점퍼로 설정한다. 버스에 있는 어떤 두 장치 사 이이든간에 동기적으로 또는 비동기적으로 데이터를 전송할 수 있으며, 32비트 크기로 초당 40 MB까지 전송할 수 있다. SCSI 버스는 장치간에 데이터와 상태 정보를 함께 전송하며, 전 송 시작자(initiator)와 전송 대상(target) 사이의 하나의 트랜잭션은 여덟개의 서로 다른 상태 를 가질 수 있다. SCSI 버스의 현재 상태는 버스에 있는 다섯개의 신호로부터 알 수 있다. 여덟개 상태는 다음과 같다.

- **버스가 비어있음(BUS FREE)** 버스에 대한 제어권을 가진 장치도, 현재 발생하는 트랜잭션도 없다.

- **중재 (Arbitration)** 한 SCSI 장치가 주소 핀에 자신의 SCSI 식별자를 내보내서 SCSI 버스에 대한 제어권을 얻어려고 하였다. 가장 높은 SCSI 식별자 번호가 이긴다.

- **선택(SELECTION)** 장치가 중재를 통해 SCSI 버스의 제어권을 얻으면, 이제 SCSI 요청을 받 을 대상에게 자신이 명령을 보내려고 한다고 신호해야 한다. 이는 주소 핀에 대상의 SCSI 식별자를 내보냄으로써 이루어진다.

- **재선택(RESELECTION)** SCSI 장치는 요청을 처리하는 도중에 연결을 끊을 수 있다. 그러면 대상은 전송 시작자를 다시 선택할 수 있다. 모든 SCSI 장치가 이 상태를 지원하는 것은 아니다.

- **명령(COMMAND)** 6, 10, 또는 12 바이트의 명령을 전송 시작자에서 전송 대상으로 전송할 수 있다.

- **데이터 입력, 데이터 출력(DATA IN, DATA OUT)** 이 상태에 있을 때에 데이터가 전송 시작 자와 전송 대상 사이에 전달된다.

- **상태(STATUS)** 모든 명령을 완료한 후에 이 상태로 들어가며, 대상이 전송 시작자에게 성공 이나 실패를 나타내는 상태 바이트를 전송할 수 있게 한다.

- **메시지 입력, 메시지 출력(MESSAGE IN, MESSAGE OUT)** 전송 시작자와 전송 대상 사이에 추가적인 정보가 전송된다.

리눅스 SCSI 서브시스템은 두가지 기본적인 요소로 되어 있다. 각각은 자료구조로 표현이 되는데 이들은 다음과 같다.

- **호스트(host)** SCSI 호스트는 하드웨어의 물리적인 부분으로, SCSI 컨트롤러이다. NCR810 PCI SCSI 컨트롤러는 SCSI 호스트의 한 예다. 리눅스 시스템이 같은 종류의 SCSI 컨트롤러 를 하나 이상 가지고 있다면, 각각은 별도의 SCSI 호스트로 표현된다. 이말은 SCSI 디바 이스 드라이버가 자신의 컨트롤러를 하나 이상 제어할 수 있다는 것이다. SCSI 호스트는 거의 항상 SCSI 명령의 전송 시작자이다.

- **장치(Device)** 가장 일반적인 SCSI 장치 종류로는 SCSI 디스크가 있지만, SCSI 표준은 테이프, CD-ROM, 그리고 일반 SCSI 장치같은 여러 종류를 더 지원한다. SCSI 장치는 거의 항상 SCSI 명령의 대상이 된다. 이들 장치는 서로 다르게 취급되어야 하는데, 예를 들어, CD- ROM이나 테이프같은 제거가능한 매체를 가진 경우 리눅스는 그 매체가 제거되었는지 인식할 수 있어야 한다. 다른 디스크 종류는 다른 메이저 장치 번호를 가지며, 이는 리눅 스가 블럭 장치에 대한 요청을 올바른 SCSI 종류로 보낼 수 있게 한다.

SCSI 서브시스템의 초기화

SCSI 서브시스템을 초기화하는 것은 SCSI 버스와 장치들의 동적인 특성 때문에 매우 복잡 하다. 리눅스는 부팅시에 SCSI 서브시스템을 초기화한다 - 시스템에 있는 SCSI 컨트롤러 (SCSI 호스트라고 하는)를 찾고, 각 SCSI 버스를 검사하여 모든 장치들을 찾아낸다. 그리고 이들 장치를 초기화하여 리눅스 커널의 다른 부분에서 일반 파일 연산과 버퍼 캐시 블럭 장 치 연산을 가지고 이들을 사용할 수 있게 한다. 초기화는 네가지 상태에서 이루어진다.

첫째, 리눅스는 커널을 빌드할 때 들어가 있는 SCSI 호스트 어댑터, 즉 컨트롤러 중 어떤 것이 자신이 제어할 하드웨어를 가지고 있는지 찾는다. 커널에 포함된 각 SCSI 호스트는 builtin_scsi_hosts 벡터에 Scsi_Host_Template 엔트리를 가지고 있다. Scsi_Host_Template 자료구조는 어떤 SCSI 장치가 SCSI 호스트에 연결되어 있는지 감 지하는 것 같은 일을 수행하는 SCSI 호스트 별로 고유한 루틴들에 대한 포인터를 가지고 있다. SCSI 서브시스템은 자신을 설정하는 동안 이들 루틴을 부르며, 이들 루틴은 이런 호스 트 유형을 지원하는 SCSI 디바이스 드라이버의 한 부분이다. 실제 SCSI 장치가 연결되어 있 는 감지된 SCSI 호스트는, 자신의 Scsi_Host_Template 자료구조를 활성화된 SCSI 호스 트들의 scsi_hosts 리스트에 추가한다. 감지된 호스트 유형의 각 호스트는 scsi_hostlist 리스트에 있는 Scsi_Host 자료구조로 표현된다. 예를 들어, 두개의 NCR810 PCI SCSI 컨트롤러를 가진 시스템은 각 컨트롤러별로 하나씩 해서, 리스트에 두개의 Scsi_Host 엔트리를 가지게 된다. 각 Scsi_Host는 자신의 디바이스 드라이버를 나타내 는 Scsi_Host_Template를 가리킨다.

이제 모든 SCSI 호스트를 발견했으므로, SCSI 서브시스템은 어떤 SCSI 장치가 호스트의 버 스에 연결되었는지 찾아야 한다. SCSI 장치는 0에서 7까지의 장치 번호를 가지는데, 이 번 호는 자신이 연결된 SCSI 버스에서 유일해야 한다. SCSI 식별자는 대개 장치에 있는 점퍼로 설정한다. SCSI 초기화 코드는 SCSI 버스에 있는 장치로 TEST_UNIT_READY 명령을 보내서 장치를 찾는다. 장치가 대답을 한다면, ENQUIRY 명령을 보내서 신원확인을 한다. 이는 리눅 스에게 제작자(vendor)의 이름과 장치의 모델명과 개정(revision) 이름을 전달한다. SCSI 명령 은 Scsi_Cmnd 자료구조로 표현되고, 디바이스 드라이버의 Scsi_Host_Template 자료구 조에 있는 디바이스 드라이버 루틴에 부를 대 이를 전달한다. 발견한 모든 SCSI 장치는 Scsi_Device 자료구조로 표현하며, 각각은 자신의 부모 Scsi_Host를 가리킨다. 모든 Scsi_Device 자료구조는 scsi_devices 리스트에 추가된다. 그림 8.4는 어떻게 이들 주 된 자료구조들이 서로 연관되어 있는지 보여준다.

SCSI 장치 유형으로는 네가지가 있다 - 디스크, 테이프, CD, 그리고 일반. 이들 SCSI 유형의 각각은 다른 메이저 블럭 장치 유형으로 커널에 개별적으로 등록된다. 그렇지만 이들은 주 어진 SCSI 장치 유형을 갖는 장치가 하나 이상 있어야 커널에 등록된다. 각 SCSI 유형은 - 예를 들어 SCSI 디스크 - 자신만의 장치 테이블을 관리한다. 이들은 이 테이블을 커널의 블 럭 연산(파일이나 버퍼캐시)을 올바른 디바이스 드라이버나 SCSI 호스트로 보내는데 사용한 다. 각 SCSI 유형은 Scsi_Device_Template 자료구조로 표현한다. 이 자료구조는 이 유 형의 SCSI 장치에 대한 정보와 다양한 작업을 수행하는 루틴들의 주소를 가지고 있다. SCSI 서브시스템은 이들 템플릿을 사용하여 각 유형의 SCSI 장치에 대해 SCSI 유형에 따른 루틴 을 부르는데 사용한다. 다르게 말하면, SCSI 서브시스템이 SCSI 디스크 장치를 연결하려고 할 때, SCSI 디스크 유형에 따른 연결 루틴을 부른다는 것이다. 어떤 유형을 갖는 SCSI 장치 가 하나 이상 발견되면 Scsi_Type_Template 자료구조가 scsi_Devicelist 리스트에 추가된다.

SCSI 서브시스템 초기화의 마지막 상태는 등록된 각 Scsi_Device_Template의 종료 (finish) 함수를 부르는 것이다. SCSI 디스크 유형이라면, 이는 발견한 모든 SCSI 디스크를 회 전시켜 그들의 디스크 구조를 기록하는 일을 한다. 또한 그림 8.3에 보이는 것처럼, 모든 SCSI 디스크를 나타내는 gendisk 자료구조를 디스크의 연결 리스크에 추가한다.

블럭 장치 요청을 전달하기

일단 리눅스가 SCSI 서브시스템을 초기화하고 나면 SCSI 장치들을 사용할 수 있게 된다. 정 상적으로 동작하는 각 SCSI 장치 유형은 자신을 커널에 등록하여, 리눅스가 블럭 장치 요청 을 자신에게 보낼 수 있게 한다. 여기에는 blk_dev를 통한 버퍼 캐시 요청이나, blkdevs 를 통하는 파일 연산이 있을 수 있다. 여기서는 하나 이상의 EXT2 파일 시스템 파티션을 가지고 있는 SCSI 디스크 드라이버를 예로 들어, 이 EXT2 파티션 중 하나를 마운트할 때 커널 버퍼 요청을 어떻게 올바른 SCSI 디스크로 전달하는지 살펴보도록 하자.

SCSI 디스크 파티션에서 한 블럭의 데이터를 읽거나 쓰는 요청은, blk_dev 벡터에 있는 SCSI 디스크의 current_request 리스트에 새로운 request 구조체를 추가하게 된다. request 리스트가 처리중이라면, 버퍼 캐시는 다른 일을 할 필요가 없다. 그렇지 않다면 SCSI 디스크 서브시스템에게 계속해서 request 큐를 처리하라고 한다. 시스템에 있는 SCSI 디스크는 Scsi_Disk 자료구조로 나타낸다. 이들은 rscsi_disks 벡터에 들어 있으 며, 이 벡터는 SCSI 디스크 파티션의 마이너 장치 번호 중 일부를 사용하여 인덱스가 되어 있다. 예를 들어 /dev/sdb1은 8번의 메이저 번호와 17번의 마이너 번호를 가지며, 이는 인덱스 1을 생성한다. 각 Scsi_Disk 자료구조는 이 장치를 나타내는 Scsi_Device 자료 구조에 대한 포인터를 갖고 있다. Scsi_Device 자료구조는 차례로 자신을 "소유"하고 있는 Scsi_Host 자료구조를 가리키고 있다. 버퍼 캐시로부터 온 request 자료구조는 SCSI 장 치로 보내야 하는 SCSI 명령을 기술하는 Scsi_Cmnd[18] 구조체로 바뀌고, 이는 이 장치를 나 타내는 Scsi_Host 구조체의 큐에 쌓인다. 한번 적절한 데이터 블럭을 읽거나 쓰고 나면, 이들은 개별 SCSI 디바이스 드라이버에 의해 처리된다.

8.6 네트웍 장치(Network Device)

리눅스의 네트웍 서브시스템은 네트웍 장치를 데이터 패킷을 보내고 받는 한 개체로 생각한 다. 이는 대개의 경우 이더넷 카드같은 물리적인 장치이다. 어떤 네트웍 장치는 소프트웨어 로만 되어 있는 것이 있는데, 데이터를 자기 자신에게 보내는데 사용되는 루프백(loopback) 장치같은 것이 그것이다. 각 네트웍 장치는 device 자료구조로 표현된다. 네트웍 디바이스 드라이버는 커널이 부팅하면서 네트웍을 초기화하는 동안 자신이 제어하는 장치를 리눅스에 등록한다[19]. 이 device 자료구조는 장치에 대한 정보와, 리눅스에서 지원하는 다양한 종류 의 네트웍 프로토콜이 장치의 서비스를 이용할 수 있도록 하는 함수들의 주소를 가지고 있 다. 이 함수들은 대부분 네트웍 장치를 통한 데이터 전송과 관계가 있다. 장치는 표준 네트 웍 지원 매커니즘을 사용하여 전송받은 데이터를 올바른 프로로콜 계층으로 전달한다. 보내 고 받는 모든 네트웍 데이터(패킷)은 sk_buff 자료구조로 표현되는데, 이는 네트웍 프로토콜 헤더를 쉽게 첨가하거나 제거할 수 있도록 만들어진 유연한 자료구조이다. 네트웍 프로 토콜 계층이 어떻게 네트웍 장치를 사용하는지, 어떻게 sk_buff 자료구조를 가지고 데이터 를 앞뒤로 전달하는지는 네트웍 장(10장)에서 상세하게 다룬다. 여기서는 device 자료 구 조와 네트웍 장치를 어떻게 발견하고 초기화하는지에 중심을 둔다.

device 자료구조는 다음과 같은 네트웍 장치에 대한 정보를 가진다.

- **이름** 특수 장치 파일이 mknod 명령으로 만들어지는 블럭 장치나 문자 장치와는 달리, 네 트웍 장치 특수 파일은 시스템에 있는 네트웍 장치를 발견하고 초기화하는 과정에서 차 례로 나타난다. 이들의 이름은 장치의 유형을 나타내는 그런 표준적인 이름이다. 같은 유 형의 장치들에는, 0부터 시작하는 번호가 붙는다. 따라서 이더넷 장치는 /dev/eth0, /dev/eth1, /dev/eth2 이런 식으로 나타난다. 일반적인 네트웍 장치로는 :

  ```
  /dev/ethN       이더넷 장치
  /dev/slN        SLIP 장치
  /dev/pppN       PPP 장치
  /dev/lo         루프백 장치
  ```

- **버스정보** 이 정보는 디바이스 드라이버가 장치를 제어하기 위해 필요로 하는 것이다. IRQ 번호는 장치가 사용하는 인터럽트 번호이고. 베이스 주소(base address)는 장치의 제어 레 지스터와 상태 레지스터가 있는 I/O 메모리 상의 주소이다. DMA 채널(DMA channel)은 네트웍 장치가 사용하는 DMA 채널 번호이다. 이 모든 정보는 부팅시에 장치를 초기화 할 때 설정된다.

- **인터페이스 플래그(Interface Flag)** 이는 네트웍 장치의 특징과 능력을 설명한다.

```
IFF_UP              인터페이스가 위에 있고(up) 실행중이다.
IFF_BROADCAST       device의 브로드캐스트 주소가 유효하다.
IFF_DEBUG           장치 디버깅 옵션이 켜져 있다.
IFF_LOOPBAK         루프백 장치이다.
IFF_POINTTOPOINT    SLIP이나 PPP같은 지점 대 지점(point to point) 연결 장치이다.
IFF_NOTRAILERS      네트웍 추적자(trailer)가 없다.
IFF_RUNNING         자원이 할당되었다.
IFF_NOARP           ARP 프로토콜을 지원하지 않는다.
IFF_PROMISC         장치가 마구잡이로 수신하는 모드이다. 패킷의 수신 주소가 어디든
                    간에 관계없이 모든 패킷을 받아 들인다.
IFF_ALLMULTI        모든 IP 멀티캐스트(multicast)[20] 프레임들을 수신한다.
IFF_MULTICAST       IP 멀티캐스트 프레임 수신 가능
```

- **프로토콜 정보** 네트웍 프로토콜 계층이 장치를 어떻게 사용할 수 있는지 나타낸다.
    - **mtu** 링크 계층(link layer)에서 붙이는 헤더를 제외하고 이 네트웍 장치가 전송할 수 있는 가장 큰 패킷 크기. 이 최대값은 IP같은 프로토콜 계층이 전송에 사용할 적당한 패킷 의 크기를 선택하기 위해 사용한다.
    - **계열(Family)** 이것은 장치가 지원할 수 있는 프로토콜 계열을 나타낸다. 모든 리눅스 네 트웍 장치가 지원하는 계열은 AF_INET, 인터넷 주소 계열이다.
    - **유형(Type)** 하드웨어 인터페이스 유형은 이 네트웍 장치에 연결된 매체를 나타낸다. 리눅스 네트웍 장치는 많은 서로 다른 종류의 매체를 지원한다. 여기에는 이더넷(ethernet), X.25, 토큰링(token ring), SLIP, PPP, 그리고 Appletalk 등이 포함된다.
    - **주소(Address)** device 자료구조는 IP 주소를 포함하여 이 네트웍 장치에 해당하는 여러 개의 주소를 가지고 있다.
- **패킷큐(Packet Queue)** 네트웍 장치가 전송하기를 기다리고 있는 sk_buff 패킷의 큐
- **지원하는 함수들** 각 장치는 장치의 링크 계층과의 인터페이스의 일부로서 프로토콜 계층에 서 호출할 수 있는 표준 함수 집합을 제공한다. 이는 셋업하고 프레임을 전송하는 루틴 뿐만 아니라, 표준 프레임 헤더를 추가하고, 통계정보를 모으는 루틴도 포함한다. 이 통 계정보는 ifconfig 명령으로 볼수 있다.

8.6.1 네트웍 장치 초기화

네트웍 디바이스 드라이버는 다른 리눅스 디바이스 드라이버와 마찬가지로 커널에 직접 포 함되어 있을 수 있다. 각 잠재적인[21] 네트웍 장치는 dev_base 리스트 포인터가 가리키는 네트웍 장치 리스트에 있는 device 자료구조로 표현된다. 네트웍 계층은 장치에 고유한 작 업을 수행할 필요가 있을 때, device 자료구조에 있는 서비스 루틴의 주소를 가지고 여러 네트웍 장치의 서비스 루틴을 호출한다. 그렇지만 device 자료구조는 처음에는 초기화나 장치를 탐사(probe)하는 루틴의 주소만 갖고 있다.

네트웍 디바이스 드라이버가 풀어야 하는 문제로 두가지가 있다. 우선 첫번째는 리눅스 커 널에 포함된 모든 네트웍 디바이스 드라이버가 자신이 제어할 장치를 갖는 것은 아니라는 것이다. 그리고 두번째로 시스템에 있는 이더넷 장치는 밑에 있는 디바이스 드라이버가 어 떤 거든지간에 항상 /dev/eth0, /dev/eth1과 같이 나타난다는 것이다. 먼저 "없는" 네트 웍 장치 문제는 쉽게 풀 수 있다. 각 네트웍 장

치의 초기화 루틴을 부르면, 이 루틴은 자신 이 구동할 컨트롤러를 찾았는지 못찾았는지 의미하는 상태값을 돌려준다. 만약 드라이버가 아무런 장치도 찾지 못했다면, dev_base가 가리키고 있는 device 리스트에 있는 엔트리 가 제거된다. 만약 드라이버가 장치를 찾게 된다면, 드라이버는 device 자료구조의 나머지 부분을 장치에 대한 정보와 네트웍 디바이스 드라이버에 있는 드라이버가 지원하는 함수들 의 주소로 채운다.

두번째 문제는 이더넷 장치를 표준 /dev/ethN 장치 특수파일에 동적으로 부여하는 문제로 이는 좀더 우아한 방법으로 해결된다. 장치 목록에는 eth0부터 eth7까지 모두 여덟개의 표준 엔트리가 있다. 초기화 루틴은 이들 모두에 똑같은데, 장치를 찾을 때까지 커널에 있는 각 이더넷 디바이스 드라이버를 시도해보는 것이다. 드라이버가 장치를 찾으면 이제 가지게 된 ethN device 자료 구조의 내용을 채운다. 그리고 이 때 네트웍 디바이스 드라이버는 자 신이 제어할 물리적인 하드웨어를 초기화하고, 어떤 IRQ를 사용하고 있고 어떤 DMA 채널 을 사용하고 있는지 (만약 있다면) 등등을 알아낸다. 드라이버는 자신이 제어할 네트웍 장치 를 여러 개를 찾을 수 있는데, 이 경우 드라이버는 여러 개의 /dev/ethN device 자료구조 을 넘겨준다. 여덟개의 표준 /dev/ethN이 모두 할당되면, 더 이상 이더넷 장치를 찾지 않 는다.

번역 : 이호, 신문석
정리 : 이호

역주 1) 이렇게 장치를 파일로 표시하는 것은 Windows 운영체제에도 영향을 미쳐, Windows 95에서는 디바이스 드라이버를 파일로 접근할 수 있으며, Windows NT 계열에서는 유닉스 와 보다 가깝게 디바이스 드라이버가 장치 파일을 만드는 형태로 되어 있다. (flyduck)

역주 2) 즉 보통 문자 장치나 블럭 장치는 실제로 장치가 존재하지 않더라도 장치 특수 파 일이 존재한다. 이는 실제 시스템에 장치가 많지 않더라도, /dev 디렉토리에 수많은 장 치 파일이 존재하는 이유이다. 하지만 네트웍 장치 파일은 실제로 장치가 존재하는 경우 에만 만들어진다. 예를 들어 시스템에 이더넷 장치가 있어야 /dev/eth0이라는 장치 특 수 파일이 생긴다. (flyduck)

역주 3) 즉 메이저 장치 번호는 디바이스 드라이버에게 부여되는 것이다. 그러므로 서로 다 른 디바이스 드라이버를 필요로 하는 CD-ROM 디바이스 드라이버는 서로 다른 메이저 번호를 가지며, 실제로 리눅스 시스템에 보면 CD-ROM 디바이스 드라이버로 여러개의 메이저 번호가 할당되어 있는 것을 볼 수 있다. 따라서 시스템에 새로운 디바이스 드라 이버를 추가할려면 사용되고 있지 않은 메이저 번호를 할당받아야 한다. 마이너 번호는 디바이스 드라이버가 자신이 관리하는 장치들을 구별하기 위해서 붙이는 것이므로, 어떤 번호를 부여하는지는 디바이스 드라이버 제작자의 몫이다. 현재 시스템에 있는 장치들의 메이저 번호와 마이너 번호의 의미는 DOCUMENTATION/Device.txt 파일에 정리되어 있다. 여기서 특이한 점은 SCSI CD-ROM이나 SCSI 디스크같은 것은 하나의 메이저 번호만을 갖는다는 것이다. 그렇다고 하나의 디바이스 드라이버가 모든 종류의 SCSI 어댑터를 지 원한다는 것은 아니다. 이는 SCSI 클래스 디바이스 드라이버가 있어서 이것이 실제로 디 바이스 드라이버를 등록하고, 각각의 SCSI 어댑터에 해당하는 디바이스 드라이버는 단지 이 SCSI 클래스 드라이버에 별도의 인터페이스를 제공하는 형태로 되어 있기 때문이다. 이는 나중에 블럭 장치에서 다시 이야기한다. (flyduck)

역주 4) 이 특성은 현재 커널이 지원하지 않는 장치가 추가되었더라도, 커널을 새로 컴파일 하지 않고 해당하는 디바이스 드라이버를 추가하여 로드함으로써 장치를 사용할 수 있게 한다. (flyduck)

역주 5) 커널을 컴파일하기 전에 make menuconfig, 또는 X 윈도우 상에서 make xconfig 명령을 통해서, 커널에 무엇을 포함하고 무엇을 모듈로 넣을 것인지 설정할 수 있다. (flyduck)

역주 6) 이는 11.3 장에서 설명하고 있는 타이머 메커니즘이다. (flyduck)

역주 7) 인터럽트 자원을 요청하는 것은 꼭 드라이버 초기화 때가 아니라도 할 수 있다. 사 람에 따라서 드라이버 초기화 때에는 어떤 인터럽트를 사용하고 있는지 확인만 하고, 실 제 인터럽트를 요청하는 것은 장치를 사용할 때에만 하며, 사용하지 않을 때는 인터럽트 자원을 반납하는 것이 좋다고 하는 사람도 있다. (flyduck)

역주 8) 프로세스는 가상 메모리를 사용하므로 할당받은 메모리는 가상 메모리 상에서는 연 속되어 있더라도 물리적으로 연속된 것은 아니다. 그래서 리눅스 커널은 DMA를 위해 특 별한 메모리 할당 함수를 제공한다. (flyduck)

역주 9) 이 "lock"의 의미는 가상 메모리가 실제 물리적으로도 존재하게 만들고, 움직여지지 않도록 만든다는 것이다. 일반적으로 메모리는 할당받더라도 물리적으로 할당받는 것이 아니기 때문에(요구 페이징), DMA에서 사용할 수 있도록 실제로 물리적으로 메모리가 존 재하게 하고 스왑 아웃되지 않게 한다는 의미이다. (flyduck)

역주 10) 이를 인터럽트 타임(interrupt time)에서 실행되고 있다고 한다. 디바이스 드라이버의 일반 서비스들은 이 서비스를 요청한 프로세스가 현재 프로세스일 때 (즉 current 가 현재 프로세스의 task_struct를 가리키고 있을 때) 실행되지만, 인터럽트 핸들러나 하 반부 핸들러, 작업큐로 처리될 때는 현재 프로세스는 전혀 상관없는 프로세스 일 수 있 다 (작업큐에서 tq_scheduler는 인터럽트 타임에서 처리되지 않는다). 그래서 이들 처 리 루틴에서는 현재 프로세스에 의존할 수 없다. 리눅스 커널에서 current는 현재 프로 세스의 task_struct를 가리킨다. (flyduck)

역주 11) 리눅스 커널은 이와 같은 페이지 기반 메모리 할당(page-oriented memory allocation) 만을 지원한다. C언어의 malloc같은 메모리 할당은 선형 메모리 할당(linear memory allocation)이라고 하는데, 리눅스 커널은 이를 지원하지 않는다. (flyduck)

역주 12) 이러한 일을 하는 시스템 콜은 register_chrdev()로, 여기에는 장치의 메이저 번호와 디바이스 드라이버의 이름, 그리고 파일 연산 블럭이 전달된다. include/linux/fs.h에 서 함수의 프로토타입(prototype)을 볼 수 있다.(flyduck)

역주 13) 이 파일 연산 블럭을 나타내는 자료구조는 file_operations로, 여기에는 open, close, read, release 같은 기본적인 연산 외에도 lseek, ioctl, fsync 등의 여러 연산들이 더 있 다. 이 자료구조는 블럭 장치의 디바이스 드라이버에도 사용된다. (flyduck)

역주 14) 이는 파일 시스템 객체에 따라서 다른 연산을 적용할 수 있게 하여, 파이프나 소 켓같이 똑같이 파일 객체 인터페이스를 가지지만 실제로 다른 동작을 하는 것을 가능하 게 한다. (flyduck)

역주 15) 디바이스 드라이버를 등록할 때 전달되는 file_operations 구조체에는 버퍼 캐 시에 관련된 함수는 없다. 그래서 블럭 장치용으로 별도의 자료구조가 필요하게 되어, 버 퍼 캐시에 관련된 blk_dev_struct 구조체와 이의 배열인 blk_dev가 존재하게 된다. 문자 장치에서와 마찬가지로 블럭 장치를 등록하는 함수인 register_blkdev()에는 메이저 번호, 이름, 그리고 파일 연산이 전달되며, 버퍼 캐시에 관련된 설정은 직접 blk_dev_struct 자료구조에 있는 request_fn 함수 포인터를 자신의 것으로 설정함 으로써 이루어진다. (flyduck)

역주 16) 한 실린더의 크기는 헤드의 수 * 섹터의 수 * 섹터 크기이므로 여기서는 64 * 32 * 512 = 1048576, 즉 1MB이다. (flyduck)

역주 17) SCSI는 여기에 디스크 외에도 스캐너같은 다른 외부장치를 붙일 수 있는 I/O 버스 규격이지만, IDE는 단지 디스크를 위한 인터페이스이다. (flyduck)

역주 18) 원문에는 Scsi_Cmd로 되어 있지만 Scsi_Cmnd가 맞다. (flyduck)

역주 19) 앞에서 설명한바와 같이 실제로 제어할 장치가 있을 때 이를 등록한다는 것이다. (flyduck)

역주 20) IP는 기본적으로 시작주소 하나와 목적지 주소 하나를 가지고 있다. 즉 어떤 곳에 서 단 하나의 목적지로만 IP 패킷을 보낼 수 있다는 것이다. 이는 화상회의같이 같은 패 킷을 여러 목적지로 보내는 경우 중복된 내용을 수신하는 숫자만큼 목적지를 따로 지정 해 보내야 하므로 많은 대역폭(bandwidth)를 잡아먹게 된다. 이에 등장한 IP 멀티캐스트는 목적지를 여러 곳을 지정할 수 있게 하여 패킷을 하나 보내면 이 패킷에 기록된 모든 목 적지로 패킷을 전송하게 해 주는 프로토콜이다. (flyduck)

역주 21) 장치가 있을 가능성은 있지만 아직 확인한 것은 아니기에 잠재적으로 존재한다. (flyduck)