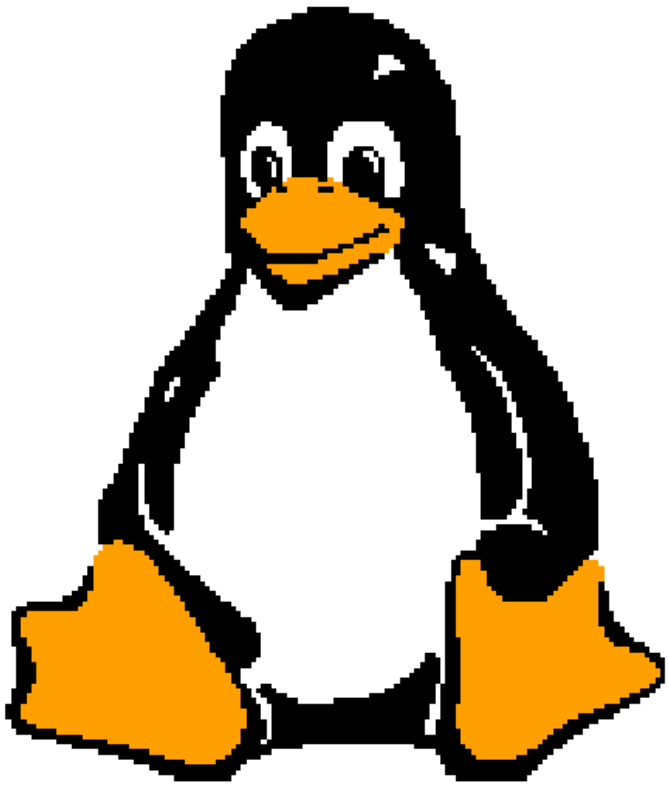


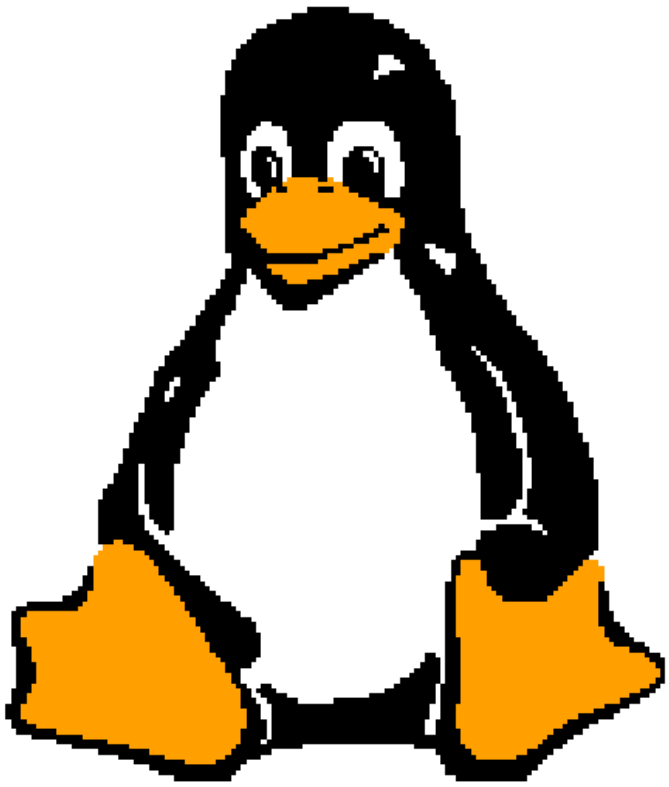
The Linux Kernel

David A Rusling
david.rusling@arm.com

REVIEW, Version 0.8-3

1999년 1월 25일

이 책은 리눅스 커널이 어떻게 동작하는지 알고 싶어하는 리눅스의 팬들을 위한 것이다. 이 책은 내부구조에 대한 매뉴얼이 아니다. 이보다는 리눅스 커널이 어떻게 동작하고 왜 그렇게 하는지, 리눅스가 사용하는 근본 원리와 작동방식을 설명한다. 리눅스는 계속 변하고 있다. 이 책은 현재 가장 안정적이고, 개인과 회사에서 가장 많이 사용하고 있는 2.0.33 소스를 바탕으로 한다.



리눅스 커널

돌도끼
linux@flyduck.com
http://linux.flyduck.com/tlk/

Version 0.8-3, 번역판 0.1.0

1999년 11월 8일

이 책은 리눅스 문서화 프로젝트(Linux Documentation Project)의 하나인 David A Rusling의 저서 <The Linux Kernel>을 번역한 것이다. 번역은 서울대 컴퓨터 연구회 졸업생 모임인 돌도끼에서 하였다. 이 문서는 자유롭게 배포할 수 있으며, 상업적으로 이용할 수 없다. 그 외는 저자가 명시한 라이선스 규약에 따른다.

Legal Notice

3 Foxglove Close, Wokingham, Berkshire RG41 3NF, UK
david.rusling@arm.com

법적 안내문

UNIX는 Univel의 등록상표이다.
리눅스는 리누스 토발즈의 등록상표이며, UNIX™나 Univel과 아무런 관련이 없다.

목차

서문

리눅스는 인터넷의 한 현상이다. 리눅스는 한 학생의 취미 프로젝트로 시작해서, 이제는 무료로 얻을 수 있는 다른 어떤 운영체제보다도 대중적으로 성장했다. 많은 사람들에게 리눅스는 수수께끼이다. 어떻게 공짜인데도 쓸만한 것일 수 있을까? 몇 안되는 대규모 소프트웨어 회사들이 판치는 세상에서, 어떻게 한 떼의 "해커들"[1]이 만든 소프트웨어가 이들과 함께 경쟁할 수 있을까? 전세계의 서로 다른 나라의 서로 다른 사람들이 기여한 소프트웨어가 어떻게 안정적이면서 동시에 효율적일 수 있을까? 하지만 리눅스는 안정적이고 효율적인 동시에 경쟁까지 펼치고 있다. 많은 대학과 연구기관에서 일상적인 컴퓨터 업무에 리눅스를 사용하고 있다. 집에 있는 컴퓨터에서 리눅스를 사용하는 사람도 있고, 대부분의 회사들도 리눅스를 사용하고 있다는 것을 실감하지는 못할지라도 어디선가 틀림없이 리눅스를 사용하고 있다. 그들은 리눅스를 웹사이트를 보고, 웹사이트를 구축하고, 이메일을 보내고, 늘 하듯이 게임을 하는데 사용한다. 리눅스는 결단코 장난감이 아니다. 리눅스는 전세계의 애호가들이 사용하고 있는, 완전히 개발되고 전문적으로 만들어진 운영체제이다.

리눅스의 기원은 Unix™의 시작까지 거슬러 올라간다. 1969년 벨 연구소(Bell Laboraties) 연구 그룹(Research Group)의 일원인 켄 톰슨(Ken Thompson)은 놀고 있는 PDP-7을 이용하여 멀티유저, 멀티태스킹 운영체제를 실험하기 시작했다. 곧 데니스 리치(Dennis Richie)가 합류하였고, 둘은 연구 그룹의 다른 사람들과 함께 유닉스의 초기 버전을 만들었다. 리치는 이전에 수행한 MULTICS라는 프로젝트에 크게 영향을 받았는데, 유닉스라는 이름 자체가 MULTICS에 빗대어 지은 것이다. 초기 버전은 어셈블리 코드로 작성하였으며, 세번째 버전은 새로운 프로그래밍 언어인 C로 다시 작성하였다. 리치는 운영체제를 만드는데 사용할 목적으로 C 프로그래밍 언어를 설계하고 만들었다. C로 고쳐썼기 때문에 디지털(Digital)의 더욱 강력한 PDP-11/45와 11/70 컴퓨터로 유닉스를 옮길 수 있었다. 그 이후는 흔히 하는 말로 역사가 되었다. 유닉스는 연구소에서 나와 컴퓨터 세계의 주류에 편입되었고, 곧 대부분의 주요 컴퓨터 생산업체들은 자신들의 유닉스 버전을 만들어 내놓았다.

리눅스는 단순한 욕구를 만족시켜준 해답이었다. 리눅스의 작성자이자 주관리자인 리누스 토발즈(Linus Tovalds)가 갖고 놀만한 유일한 소프트웨어는 미닉스(Minix)였다. 미닉스는 유닉스와 비슷한, 간단한 운영체제로서 교육목적으로 널리 쓰이고 있었다. 리누스는 미닉스의 기능에 만족하지 못했고, 자기나름의 소프트웨어를 만들어 이를 해결하려고 했다. 그는 학창시절 때 익숙한 운영체제인 유닉스를 모델로 삼고, 인텔 386 PC에서 프로그램을 만들기 시작했다. 작업은 매우 빠르게 진척되었고, 이에 고무 받은 리누스는 자신의 노력의 결과물을 당시 막 등장하던 전세계적인 컴퓨터 네트웍을 통하여 다른 학생들에게 제공하였다. 그리하여 리눅스는 대학 사회에서 주로 쓰이게 되었다. 소프트웨어를 보고 이에 공헌하는 사람이 나타나기 시작했다. 그들은 자신이 이전에 가졌던 문제점들을 해결하는데 사용했던 방법들을 새로운 소프트웨어에 적용했다. 오래지 않아 리눅스는 운영체제의 모습을 갖추게 되었다. 중요한 점은 리눅스가 유닉스 코드를 단 한 줄도 가지고 있지 않다는 사실이다. 리눅스는 공표된 POSIX 표준에 따르는 완전히 새로 짜여진 것이다. 리눅스는 메사추세츠주 캠브리지에 있는 무료 소프트웨어 재단(Free Software Foundation)에서 만든 GNU (GNU's Not Unix™) 소프트웨어로 만들어지고, 많은 GNU 소프트웨어들을 사용한다.

대부분의 사람들은 단지 여러 종류의 잘 만들어진 CD-ROM 배포판 중 하나를 설치하고, 리눅스를 간단한 도구로서 사용한다. 상당수의 리눅스 사용자들은 프로그램을 작성하거나, 다른 사람이 만든 프로그램을 실행하는데 리눅스를 사용한다. 많은 리눅스 사용자들은 열심히 HOWTO[2] 문서를 읽고, 시스템의 한 부분을 제대로 설정하였을 때는 성공의 전율감을 느끼지만, 동시에 제대로 동작하지 않는 경우 실패의 좌절감을 맛보기도 한다. 소수의 사용자는 디바이스 드라이버를 만들고, 커널을 수정하여 리눅스 커널의 제작자이면서 관리자인 리누

역주 1) 원문에는 "hackers" (sic)라고 하고 있는데, sic는 "원문대로"라는 의미로 의심나는 원문을 그대로 인용할 때 쓰는 표기이다. (flyduck)

2) HOWTO는 말그대로 무언가를 어떻게 하면 되는지 적어놓은 문서이다. 많은 HOWTO 문서들이 리눅스 용으로 존재하며, 이들 모두 대단히 유용하다.

스 토발즈에게 보낼 정도로 열성이다. 리누스는 어디서든, 누구한테서든 커널 소스에 대한 추가나 수정을 받아들인다. 이것은 얼핏 무정부주의적인 방법처럼 들릴 수도 있겠지만, 리누스는 새 코드들을 엄격하게 검사하고 모든 코드를 자신이 직접 커널에 추가한다. 그렇긴 하지만, 실제로 어느 한 시점에서 리눅스 커널 소스 작업에 참여하는 사람들의 수는 손에 꼽을 수 있는 정도에 불과하다.

대부분의 리눅스 사용자들은 운영체제가 어떻게 동작하는지, 각 부분들이 어떻게 맞물려 돌아가는지 눈여겨 보지 않는다. 그러나 리눅스를 자세히 살펴보는 것은 운영체제의 동작 원리를 배우는 훌륭한 방법임을 생각해 본다면 이는 부끄러운 일이다. 리눅스 소스는 매우 잘 만들어졌을 뿐만 아니라 그 누구든지 자유롭게 살펴볼 수 있도록 완전히 공개되어 있다. 이것은 작성자들이 소프트웨어에 대해 저작권을 소유하고는 있지만, 무료 소프트웨어 재단의 GNU 공개 라이선스에 따라 그 소스 코드를 자유롭게 배포할 수 있도록 하기 때문이다. 그렇지만 막상 소스 코드를 처음 마주칠 때는 이들이 뭔지 무척 혼동스러울 것이다. kernel, mm, net 등의 디렉토리가 눈에 띄겠지만, 이 안에는 어떤 코드가 담겨 있으며 그 코드들은 어떻게 동작할 것인가? 여기서 필요한 것은 리눅스의 전반적인 구조와 목적에 대해 폭넓게 이해하는 것이다. 이것은 간단히 말하자면 바로 이 책의 목적이기도 하다. 즉 리눅스라는 운영체제가 어떻게 동작하는지 명확히 이해할 수 있도록 도와주는 것이다. 다시 말하면 파일을 복사하거나 이메일을 읽을 때, 리눅스 시스템 내부에서 어떤 일들이 일어나는 가를 머릿속에서 그려볼 수 있도록 하려는 것이다. 필자는 운영체제가 실제로 어떻게 동작하는지 처음 깨달았을 때 느꼈던 흥분을 아직도 잘 기억하고 있다. 그 흥분이 이 책을 읽는 독자들에게 전해지길 바란다.

내가 처음으로 리눅스와 관계를 맺은 것은 1994년 후반, 알파 AXP 프로세서 기반 시스템으로 리눅스 포팅 작업을 하던 짐 파라디스(Jim Paradis)를 방문했을 때다. 나는 1984년부터 디지털 이퀴먼트(Digital Equipment Co, Ltd, 줄여서 DEC 또는 디지털이라고 함)에서 주로 네트웍 및 통신관련 분야에서 일해 왔었고, 1992년에는 새로 구성된 디지털 반도체 부서에서 업무를 맡게 되었다. 그 부서의 목표는 상용 반도체칩 공급 시장에 전면적으로 진입하여 칩을 판매하는 것으로, 구체적으로는 알파 AXP 계통의 마이크로프로세서와, DEC 외부에서 설계한 알파 AXP 시스템 보드까지도 취급, 판매하는 것이었다. 처음 리눅스에 대해 듣고, 나는 금방 리눅스에 흥미를 갖게 되었다. 나는 짐의 열정에 끌렸고, 포팅 작업을 돕기 시작했다. 일을 계속해 가면서 나는 리눅스 운영체제 자체뿐만 아니라 그것을 만들어 내는 엔지니어들의 공동체에 대해서도 점점 그 진가를 알게 되었다.

그러나 알파 AXP는 리눅스가 동작하는 수많은 하드웨어 플랫폼 가운데 하나에 지나지 않는다. 현재 리눅스 커널의 대부분은 인텔 프로세서 기반 시스템에서 돌고 있지만 비인텔 리눅스 시스템의 수도 점점 늘어나고 있는 추세다. 알파 AXP, ARM, MIPS, Sparc, 그리고 PowerPC 등이 그 예다. 나는 이 책을 쓰면서 이들 플랫폼 중에서 어떤 것이든 이용할 수 있었지만, 나의 리눅스에 대한 기반지식과 기술적인 경험은 주로 알파 AXP에서이고, ARM에서 도는 리눅스에 대해 어느정도 알고 있는 것이다. 이 책에서 종종 키 포인트를 설명하는데 비인텔 하드웨어를 예로 드는 것은 이 때문이다. 리눅스 커널 소스의 95% 가량은 리눅스가 동작하는 모든 하드웨어에서 그대로 사용되고 있다. 마찬가지로 이 책의 95% 가량은 리눅스 커널 중에서 하드웨어와 무관한 부분에 대한 것이다.

독자의 자세

이 책은 독자가 가진 지식이나 경험에 대해 아무런 가정도 하지 않고 있다. 나는 독자가 주제로 다루는 문제에 관심을 가지고 있다면, 필요한 부분에 대해서 스스로 공부하게 되리라고 믿는다. 컴퓨터에 친숙하고(아마도 PC이겠지만), C 프로그래밍 언어에 대해 어느정도 지식을 가진다면, 독자 여러분이 이 글에서 실질적인 이득을 얻는데 도움이 될 것이다.

이 책의 구조

이 책은 리눅스의 내부구조에 대한 매뉴얼로 만든 것이 아니다. 그 대신 일반적인 운영체제, 특히 리눅스에 대한 소개서가 될 것이다. 각 장(□)은 "일반적인 것에서 시작하여 특수한 것으로 작업하는" 나의 규칙을 따르고 있다. 먼저 그 장에서 설명하려는 커널 서브시스템의 개요를 제시하고 나서, 다음으로 끔찍한 상세내용들을 다룬다.

나는 일부러 커널의 작동방식인 알고리즘을, 함수 X()가 어떤 자료구조의 어떤 항목의 값을 증가시키는 함수 Y()를 부른다는 식으로 설명하지 않았다. 이런 것들은 코드를 읽어보면 알 수 있는 것이다. 나는 어떤 코드를 이해하거나 다른 사람에게 그것을 설명해야 할 때마다, 종종 칠판에 자료구조를 그리는 일부터 시작하곤 했다. 마찬가지로, 나는 여러개의 서로 관련된 커널 자료구조와 그들의 상관관계를 아주 자세하게 설명했다.

각 장에서 다루는 리눅스 커널 서브시스템이 그렇듯, 이들을 다루는 각 장은 아주 독립적이다. 하지만 이따금 몇몇 장은 연관되어 있다. 예를 들면, 가상 메모리가 어떻게 동작하는지 이해하지 못한 채 프로세스를 설명할 수는 없을 것이다.

1장 "하드웨어의 기초" 장에서는 요즘의 PC에 대해 간략히 소개한다. 운영체제는 자신의 토대가 되는 하드웨어 시스템과 긴밀하게 연결되어 동작해야 하며, 하드웨어만 제공할 수 있는 몇몇 서비스들을 필요로 한다. 리눅스 운영체제를 완전히 이해하려면 관련된 하드웨어의 기본적인 것들을 이해해야 한다.

2장 "소프트웨어의 기초" 장에서는 기본적인 소프트웨어 원칙을 소개하고, 어셈블리와 C 프로그래밍 언어를 살펴본다. 이 장에서 리눅스와 같은 운영체제를 만드는데 사용하는 툴을 살펴보고, 운영체제의 목적과 기능들을 간략히 소개한다.

3장 "메모리 관리" 장에서는 리눅스에서 시스템상의 실제 메모리와 가상 메모리를 관리하는 방법을 설명한다.

4장 "프로세스"장에서는 프로세스란 무엇이며, 리눅스 커널이 어떻게 프로세스를 생성하고 관리하며 삭제하는가에 대하여 설명한다.

프로세스는 그들의 활동을 통합하기 위해 프로세스 사이에, 그리고 커널과 통신한다. 리눅스는 여러 종류의 프로세스간 통신(Inter-Process Communication, IPC) 구조를 지원한다. 시그널과 파이프는 이들 중 일부이며, System V IPC(이것이 처음 등장한 유닉스 버전의 이름을 따라 붙여진 이름이다) 메커니즘 역시 지원한다. 이러한 프로세스간 통신 메커니즘은 5장에서 설명하고 있다.

PCI(Peripheral Component Interconnect) 표준은 이제 저비용 고성능 PC용 데이터 버스로서 확고하게 자리잡았다. 6장 "PCI"장에서는 리눅스 커널이 시스템 내의 PCI 버스들과 장치들을 초기화하고 사용하는 방법을 설명한다.

7장 "인터럽트와 인터럽트 처리" 장에서는 리눅스 커널이 어떻게 인터럽트를 다루는지 살펴본다. 커널이 인터럽트를 처리하는 데에는 일반적인 메커니즘과 인터페이스가 있지만, 세부적인 인터럽트 처리는 하드웨어와 아키텍처에 따라 다르다.

리눅스의 강점중의 하나는 요즘 PC에서 사용할 수 있는 많은 하드웨어 장치들을 지원한다는 것이다. 8장 "디바이스 드라이버" 장에서는 리눅스 커널이 시스템에 있는 물리적인 장치를 제어하는 방법을 설명한다.

9장 "파일 시스템" 장에서는 리눅스 커널이 어떻게 파일 시스템 내의 파일들을 다루는지 설명한다. 또한, 가상 파일 시스템(Virtual File System, VFS)과 리눅스 커널의 실제 파일 시스템 지원 방법도 설명한다.

네트워킹과 리눅스는 거의 같은 의미를 가지는 단어이다. 실제로 리눅스는 인터넷, 즉 월드 와이드 웹(World Wide Web, WWW)의 산물이다. 리눅스 개발자들과 사용자들은 웹을 이용하여 정보와 아이디어, 코드를 교환하며, 리눅스 자체는 종종 단체들의 네트워킹에 대한 요구를 지원하기 위해 사용된다. 10장 "네트워크" 장에서는 어떻게 리눅스가 알려진 네트워크 프로토콜을 지원하는지 TCP/IP로 총괄하여 설명한다.

11장 "커널 메커니즘" 장에서는, 커널의 여러 부분들이 효율적으로 함께 동작할 수 있도록 리눅스 커널이 제공하는 몇가지 일반적인 작업과 메커니즘에 대해 살펴본다.

12장 "모듈" 장에서는 리눅스가 어떻게 파일 시스템같은 기능요소들을 동적으로, 필요로 할 때에만 로드할 수 있는지 설명한다.

13장 "프로세서" 장은 리눅스가 포팅되어 있는 여러 프로세서들에 대한 간략한 소개글을 담고 있다.

14장 "소스" 장은 커널의 특정 기능에 대해 알고자 할 때, 리눅스 커널 소스 코드 어느곳부터 시작해야 하는지 설명한다.

본서의 표기법

이 책에서 사용한 활자체 표기법은 다음과 같다.

serif 글꼴 독자가 직접 그대로 입력해야 하는 명령어 또는 문장을 의미한다.
type 글꼴 자료구조나 자료구조내의 항목을 가리킨다.

foo/bar.c 의 foo() 참조

책 전체에 걸쳐서 리눅스 커널 소스에 있는 코드에 대한 참조 표시가 있다 (그 예로, 본 문장에 인접해 있는 테두리가 있는 글상자). 이는 독자가 소스 코드 자체를 살펴보고자 할 경우를 위해서이며, 참조하는 모든 파일은 `/usr/src/linux` 디렉토리를 기준으로 한 상대위치이다. 예를 들어 `foo/bar.c` 파일의 경우 완전한 파일명은 `/usr/src/linux/foo/bar.c`가 될 것이다. 현재 리눅스를 실행중이라면 (당연히 그래야 하겠지만) 소스 코드를 들여다보는 것은 가치있는 경험이며, 독자는 이 책을 소스 코드의 이해를 돕고 또 여러 자료구조의 의미를 파악하는 데 유용한 지침서로 사용할 수 있을 것이다.

등록상표

저자 소개

나는 1957년, 스푸트니크호가 발사되기 몇 주전 영국 북부에서 태어났다. 나는 대학에서 유닉스를 처음 접했다. 거기서 강사는 유닉스를 커널과 스케쥴링 및 다른 운영체제의 개념들을 가르칠 때 예로 사용했다. 나는 졸업년도 프로젝트를 위해 도입된 PDP-11 시스템을 즐겨 사용했다. 1982년 컴퓨터과학과를 최우등으로 졸업한 뒤, 나는 Prime Computers (Primos)에서 근무하였고, 2년 후 디지털(Digital)로 옮겼다(VMS, Ultrix). 디지털에 재직하는 동안 다양한 업무를 맡아 일했는데, 마지막 5년간은 알파 및 StrongARM 평가용 보드를 설계하는 반도체 그룹에서 일했다. 1998년 ARM으로 옮겨 로우 레벨 펌웨어를 만들고 운영체제를 포팅하는 작은 엔지니어 그룹을 맡게 되었다. 내 아이들(에스더와 스티븐)은 아빠를 괴짜라고 부른다.

사람들이 종종 직장에서나 집에서 리눅스에 대한 질문을 던지는데, 나는 그저 고맙고 행복할 뿐이다. 직업상으로 또 개인적으로 리눅스를 쓰면 쓸수록, 나는 점점 더 리눅스 광신도(zealot)가 되어가고 있다. 독자들은 여기서 맹신도(bigot) 등이 아닌 광신도(zealot)라는 용어를 썼음을 눈여겨 보아주기 바란다. 필자는 '리눅스 광신도'를, 다른 운영체제들의 존재 역시 인식하고 있지만 안쓰는 편을 택한 열성분자로 정의한다. 윈도우즈 95를 쓰는 나의 아내 길(Gill)이 언젠가 이렇게 말했다. "우리가 남편 운영체제니 아내 운영체제니 하는 말을 쓰게 될거라곤 짐작도 못했어요". 엔지니어인 내게 있어서 리눅스는 나의 요구에 완벽하게 맞아떨어진다. 리눅스는 내가 집과 회사에서 같이 사용할 수 있는 유연하고 적용하기 쉬운 엔지니어링 도구이며 최고의 운영체제이다. 공짜로 사용할 수 있는 소프트웨어의 대부분은 리눅스에서 쉽게 컴파일이 되며, 때로는 미리 컴파일 된 실행파일을 다운로드 받거나 CDROM에서 설치할 수도 있다. 공짜로 C++이나 Perl 프로그래밍을 배우고, Java에 대해 공부하는데 사용할 수 있는 다른게 무엇이 있는가!

감사의 글

우선 시간을 내어 이 책에 대해 이메일로 주석을 달아 보내준 많은 분들에게 감사한다. 나는 새 판을 낼 때마다 이들 주석들을 모두 포함시키려고 해 왔으며, 이는 주석을 받는 것보다 더 행복한 작업이었다. 어쨌든 나의 새 이메일 주소를 기억해주면 고맙겠다.

많은 강사들이 이 책의 일부를 컴퓨터를 가르치는데 쓸 수 있는지 편지로 물어왔다. 이에 대한 나의 대답은 당연히 "예"이다. 이는 내가 특히 바랬던 이 책의 용도 중 하나이다. 그 수업받는 학생들 중에 또 다른 리누스 토발즈가 앉아있을지 누가 알겠는가.

책 전반에 관해서 상세하게 검토해 준 존 릭비(John Rigby)와 마이클 바우어(Miachel Bauer)에게 특별히 감사드린다. 쉬운 일은 아니었을텐데도 나의 질문에 참을성있게 대답을 해준 앨런 콕스(Alan Cox)와 스티븐 트위디(Stephen Tweedie)에게도 감사드린다. 각 장을 좀더 즐겁게 하기 위해 래리 에윙(Larry Ewing)의 펭귄을 사용했다. 끝으로, 이 책을 리눅스 문서화 프로젝트(Linux Documentation Project, LDP)로 받아주고 웹사이트에 올려준 그렉 핸킨스(Greg Hankins)에게 감사드린다.

역자 서문

이 책은 리눅스 문서화 프로젝트의(Linux Documentation Project)의 하나인 David A Rusling의 저서 <The Linux Kernel>을 바탕으로 번역한 것이다. 번역은 서울대 컴퓨터 연구회 졸업생 모임인 돌도끼의 많은 사람들이 참여하여 이루어졌다. 원래는 돌도끼 내의 리눅스 연구 모임에서 리눅스 커널을 공부하던 중, 이 글이 리눅스 커널을 이해하기에 좋은 글이라는 생각이 들어 번역을 시작하게 되었고 많은 사람들의 도움을 받아서 이루어지게 되었다.

번역은 1차 번역과 2차 번역 두 과정으로 이루어졌으며, 되도록 원문을 뜻을 살리면서 문장을 알기 쉽게 가다듬고, 문맥을 자연스럽게 하려고 했다. 그리고 설명이 필요한 부분에 대해서는 되도록 많은 주석을 달려고 했으며, 책의 내용에 추가할 내용은 각 장의 끝에 추가하려고 했다.

원문은 Version 0.8-3을 바탕으로 하였고, 책에 있는 감수 노트(REVIEW NOTE) 등도 그대로 놔두었다. 번역판의 첫 버전은 0.1.0이고, 여기에는 서문을 포함하여 1장부터 14장까지의 원문의 내용이 들어 있다. 여기에는 부록은 빠져 있으며, 마지막에 용례집이 들어있다.

이 글이 모든 것을 담고 있고 완전한 것은 아니지만, 리눅스를 공부하는 사람들에게 많은 도움이 되리라고 생각한다. 이 글에 잘못된 부분이 있거나, 보충할 내용이 있으면 언제든지 linux@flyduck.com으로 메일을 보내주시고, 다음에 개정판을 낼 때 반영하도록 하겠습니다. 이 글의 가장 최신판은 `http://linux.flyduck.com/tlk/`에서 구할 수 있다.

참고로 아직 이 번역에 대해 저자에게서 공식적인 허락을 받진 못했다. 저자에게 편지로 이 문서를 한국어로 번역한다고 이야기를 했지만 답장이 오지 않았기 때문에, 앞의 안내문에 따라 암묵적으로 허가한 것으로 생각하고 있다.

마지막으로 좋은 글을 쓴 David A Rusling과 이 번역과정에 참여해 준 모든 분들께 감사드린다.

1999년 11월 8일
돌도끼

번역 기록

1차 번역 : 1999년 7월 30일 - 1999년 9월 17일

1차번역은 게시판을 통해 사람들이 자유롭게 참여할 수 있는 형태로 이루어졌다. 여기에는 스물명이 넘는 사람들이 참여하였지만, 다음 네 사람이 큰 공헌을 하였다.

이호 (flyduck) : flyduck@flyduck.com
심마로 (maro) :
고양우 (cat) :
김성룡 :

이외에 다음 사람들이 많은 도움을 주었다.

서창배 (cbsuh)
신문석 (scmoon)
김기용 (gyong)
김진석 (jinsuk)
손은석 (soneus)
정직한 (honest)
윤경일 (kiyoon)
홍경선 (liberty)
이승 (icarus)
이승철 (sclee)
이대현 (donky)

그 밖에 참여한 사람들은 다음과 같다.

이준희 (jhlee), 황태연 (dolphin), 김현석, truejaws, PCK, 김종원, 금화섭, 홍석근

2차 번역 : 1999년 9월 18일 - 1999년 11월 7일

2차 번역에는 다음 세사람이 참여하였으며, 각각 다음과 같은 부분을 맡았다. 2차 번역에는 오역을 잡고, 문맥을 다듬으며, 주석을 다는 일이 포함되었으며, 최종적으로 그림을 추가하고 문서 포맷을 일치시키는 일이 추가되었다.

이호 (flyduck) : 1장 - 8장, 11장 - 14장
고양우 (cat) : 9장
심마로 (maro) : 10장

Version 0.8-3, 번역판 0.1.0 : 1999년 11월 8일

서문과 1장부터 14장, 용례, 간단한 목차를 포함하고 있다.

1장

하드웨어의 기초

운영체제는 그의 기반이 되는 하드웨어 시스템과 밀접한 관계를 가지고 동작해야 한다. 운영체제는 하드웨어만이 제공할 수 있는 특정 서비스들을 필요로 한다. 리눅스 운영체제를 완전히 이해하려면 이의 기반이 되는 하드웨어의 기본 사항들을 이해하고 있어야 한다. 이 장에서는 하드웨어 - 요즘의 PC - 에 대해 간단히 소개하도록 하겠다.

"Popular Electronics" 잡지의 1975년 1월호 표지에 알테어(Altair) 8080의 삽화가 등장했을 때부터 혁명은 시작되었다. 스타트렉 초기 에피소드에 등장하는 목적지의 명칭을 따서 이름지어진 알테어 8080은[3], 취미로 전자 공작을 즐기는 열성파들이 겨우 397 달러만 들이면 조립할 수 있는 것이었다. 인텔 8080 프로세서와 256 바이트의 메모리에 화면과 키보드도 없어 요즘 기준으로는 보면 보잘것 없는 것이다. 이것을 개발한 에드 로버트(Ed Roberts)는 자신의 새 발명품에 "개인용 컴퓨터(personal computer, PC)"라는 이름을 붙였는데, 이제 이 PC라는 용어는 혼자서 들 수 있는 크기의 대부분의 컴퓨터를 가리키게 되었다. 이 정의에 따르면 매우 강력한 성능을 발휘하는 알파 AXP 시스템 역시 PC라고 할 수 있다.

열렬한 해커들은 알테어의 잠재력을 알아보았고, 이를 위한 소프트웨어를 작성하고, 하드웨어를 제작하기 시작했다. 그것은 이들 초기 선구자들에게 있어 자유 - 엘리트 성직자에 의해 실행되고 보호되는 거대한 일괄처리 메인프레임 시스템으로부터의 자유 - 를 의미했다. 자기집 식탁 위에 놓을 수 있는 컴퓨터라는 이 새로운 현상에 고무된 대학 중퇴자들은 순식간에 큰 돈을 벌게 되었다. 조금씩 다른 수많은 하드웨어가 등장했고, 소프트웨어 해커들은 이 새로운 기계용으로 소프트웨어를 만들 수 있어서 행복했다. 역설적이게도 요즘의 PC 형태의 기반을 만든 것은 1981 년 IBM PC를 발표하고 1982년 초 이를 고객들에게 판매하기 시작한 IBM이었다. 인텔 8088 프로세서에 64K 메모리 (256K까지 확장가능했다), 두 개의 플로피 디스크 드라이브와 가로 80글자, 세로 25줄의 문자를 표시할 수 있는 CGA (Color Graphic Adapter)[4] 카드를 장착한 이 컴퓨터는 요즘 기준으로 본다면 별로 강력하진 않지만 매우 잘 팔렸다. 이를 이어 IBM은 1983년, 당시엔 사치품으로 여겨진 10M 바이트의 용량의 하드 디스크가 달린 IBM-XT를 내놓았다. 오래지않아 컴팩(Compaq)을 포함한 많은 회사들이 IBM PC를 모방한 컴퓨터들을 생산하게 시작했고, 이 PC의 구조는 사실상 표준이 되었다. 이 실질적인 표준은 수많은 하드웨어 업체들이 성장단계의 시장을 놓고 경쟁하게 만들었고, 이로 인해 낮아진 가격에 고객들은 좋아했다. 이 초창기 PC가 가진 시스템 구조적 특징 중 많은 것들이 지금의 PC에까지 그대로 이어져 왔다. 예를들어, 가장 강력한 인텔 펜티엄 CPU를 채용한 시스템조차도, 처음 시작할 때 인텔 8086의 어드레싱 모드[5]에서 시작한다. 리

역주 3) Altair는 독수리자리의 알파별의 이름으로 우리말로 견우성이라고 한다. (jhlee)

역주 4) IBM PC 초창기에 사용했던 컬러 그래픽 카드 (flyduck)

역주 5) 인텔 8086 CPU는 모두 1M 바이트를 나타낼 수 있는 20비트 어드레싱 모드에서 동작하며, 인텔 80386 이후의 CPU는 (펜티엄을 포함하여) 32비트 어드레싱 모드에서 4G 바이트까지 메모리를 사용할 수 있지만 처음 시작할 때는 8086과 마찬가지로 20비트 어드

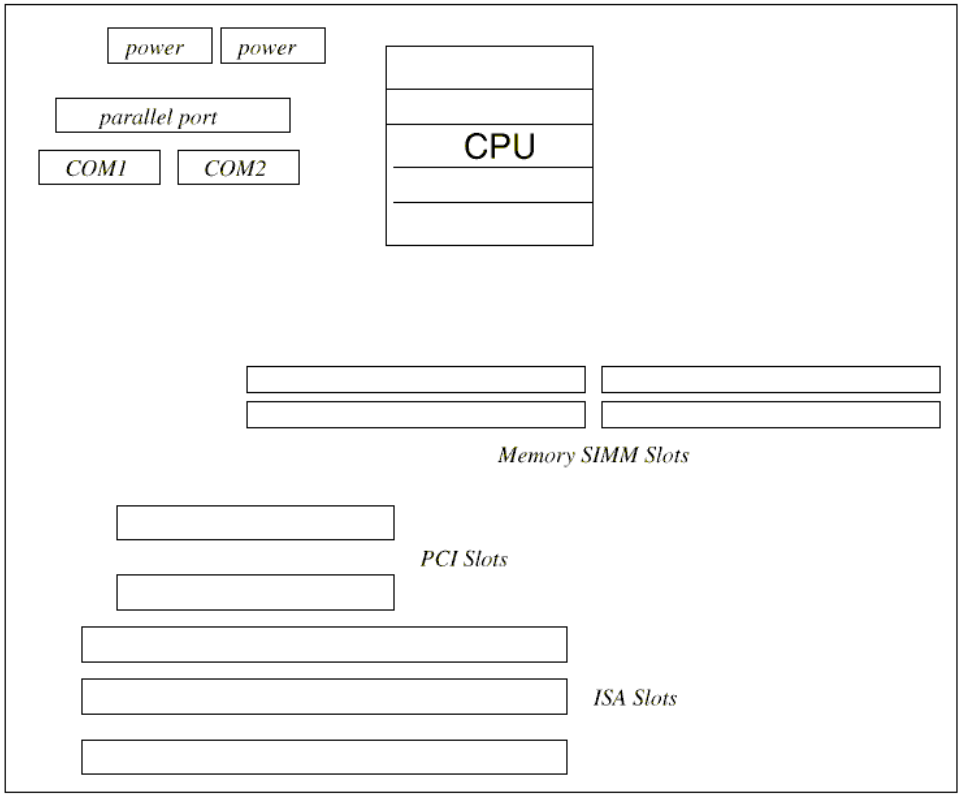


그림 1.1 : 전형적인 PC 마더보드

누스 토발즈가 나중에 리눅스라고 불리게 된 프로그램을 짜기 시작했을 때, 그는 당시 가장 널리 보급되어 있었고, 가격도 적당한 하드웨어였던 인텔 80386 PC를 선택했다.

PC의 외관을 보면, 가장 분명하게 구분할 수 있는 것은 시스템 박스와 키보드, 마우스, 그리고 모니터이다. 시스템 박스의 앞면에는 몇 개의 버튼과, 숫자를 보여주는 작은 디스플레이[6], 그리고 플로피 드라이브가 있다. 요즘에 나온 대부분의 시스템에는 CD ROM이 달려있고, 데이터 보호를 필요로 하는 경우 백업용 테입 드라이브도 있을 것이다. 이들 장치들을 총괄하여 주변장치라고 한다.

CPU가 시스템 전체를 통제하긴 하지만, CPU만이 시스템에서 지능을 가진 유일한 장치는 아니다. IDE 컨트롤러 같은 주변장치 컨트롤러 모두 어느정도 수준의 지능을 가지고 있다. PC 내부에는 (그림 1.1) CPU(또는 마이크로프로세서라고 한다)와 메모리, 여러개의 ISA나 PCI 주변장치 컨트롤러를 꽂을 수 있는 슬롯을 갖춘 마더보드가 있다. IDE 디스크 컨트롤러같은 몇몇 컨트롤러는 시스템 보드상에 있기도 하다.

1.1 CPU

CPU(CPU보다는 마이크로프로세서란 이름이 더 적당하다)는 모든 컴퓨터 시스템의 핵심이다. 마이크로프로세서는 메모리에서 명령을 읽고 이를 수행함으로써, 계산을 하고 논리 연산을 수행하고, 데이터 흐름을 관리한다. 컴퓨터가 등장한 초창기에는 마이크로프로세서의 이런 기능들이 각각 별도의 장치로 (실제로 큰 덩치의 장치로) 되어 있었다. 이 때는 중앙처리장치(Central Processing Unit, CPU)라는 말이 적합했다. 지금의 마이크로프로세서는 이들 기능

레싱 모드에서 시작한다. (flyduck)

역주 6) 486 이전의 케이스에는 터보 모드를 위하여 클럭 속도를 보여주는 LED가 달려 있었는데, 요즘 PC에는 터보 모드라는것이 없기 때문에 요즘에 나오는 케이스에는 달려있지 않다. (flyduck)

요소들을 결합해 매우 작은 실리콘 조각 하나에 집적회로로 가지고 있다. 이 책에서는 CPU, 마이크로프로세서(microprocessor), 프로세서(processor)라는 용어를 모두 같은 의미로 사용한다.

마이크로프로세서는 0과 1의 결합인 이진 데이터로 동작한다. 이 0과 1은 꺼진 상태와 켜진 상태를 갖는 전기스위치와 같은 것이다. 십진수로 42가 10짜리 4개와 1짜리 2개를 의미하는 것처럼, 이진수는 각각의 이진 숫자가 2의 몇제곱승을 나타내는 2진 숫자의 연속이다. 여기서 몇제곱승이란 같은 숫자를 여러번 곱하는 횟수를 말한다. 10의 1 제곱승(10^1)은 10이고, 10의 2제곱승(10^2)은 10x10, 10^3은 10x10x10 등등이다. 이진수 0001은 십진수로 1, 이진수 0010은 십진수 2, 이진수 0011은 십진수 3, 이진수 0100은 십진수 4에 해당한다. 따라서 십진수 42는 이진수로 101010, 즉 $2 + 8 + 32$ 또는 $2^1 + 2^3 + 2^5$이다. 컴퓨터 프로그램에서는 일반적으로 숫자를 나타내는데 이진수를 쓰기 보다는 다른 진법인 십육진수를 사용한다. 십육진법에서는 각 숫자가 16의 몇제곱승을 나타낸다. 숫자는 0부터 9까지만 있으므로 10부터 15까지는 문자 A, B, C, D, E, F로 표시한다. 예를들어 십육진수 E는 십진수로 14이고, 십육진수 2A는 숫자 42(16짜리 2개 + 10)이 된다. C 프로그래밍 언어에서는 십육진수 앞에 "0x"를 붙여서 구별한다. 즉 십육진수 2A는 0x2A라고 쓴다. 이 책에서는 이 표기법을 사용한다.

마이크로프로세서는 덧셋, 곱셈, 나눗셈 같은 숫자 연산과 "X가 Y보다 큰가?"같은 논리 연산을 수행할 수 있다.

프로세서의 명령 수행은 외부 클럭에 의해 제어된다. 이 클럭을 시스템 클럭이라고 하며, 정기적으로 클럭 펄스를 만들어 프로세서로 보내고, 각 클럭 펄스마다 프로세서는 주어진 일을 하게 된다. 예를 들어, 어떤 프로세서는 각 클럭 펄스마다 명령어를 하나씩 처리한다. 프로세서의 속도는 초당 시스템 클럭의 횟수로 나타내는데, 예를 들어 100MHz 프로세서는 초당 1억번의 클럭 틱을 받는다. 그러나 프로세서마다 한번의 클럭 틱 동안 수행하는 일의 양이 다르기 때문에, CPU의 성능을 클럭 속도로 비교하는 것은 잘못된 것이다. 하지만 모든 점들이 똑같다면, 클럭 속도가 빠른 것이 더 강력한 프로세서이다. 프로세서가 수행하는 명령은 매우 단순한 것이다. 예를 들면 "메모리 X 위치에 있는 내용을 레지스터 Y로 읽어들여라" 같은 것이다. 레지스터(register)는 데이터를 저장하고 연산을 하는데 사용하는 마이크로프로세서 내부에 있는 기억장소이다. 어떤 명령은 프로세서가 하던 일을 중단하고 메모리의 다른 위치에 있는 또 다른 명령어로 건너뛰게 하기도 한다. 이런 자그만 명령 단위는 프로세서가 1초에 수백만에서 심지어 수십억개의 명령어를 실행할 수 있게 하여, 지금의 프로세서가 거의 무한한 능력을 가질 수 있게 한다.

명령어를 수행하려면 먼저 명령어를 메모리에서 가져와야 한다. 어떤 명령어는 메모리에 있는 데이터를 참조하기도 하는데, 이 경우 메모리에서 데이터를 가져와야 하며, 데이터를 쓰려고 하는 경우 메모리에 데이터를 저장하게 된다.

프로세서에 있는 레지스터의 크기와 갯수, 종류는 프로세서 종류마다 다르다. 인텔 486 프로세서는 알파 AXP 프로세서와 다른 레지스터 세트를 가진다. 우선 인텔의 레지스터는 32비트 크기지만 알파 AXP의 레지스터는 64비트이다. 그렇지만 대체로 어떤 프로세서이든 여러개의 일반 목적 레지스터와 이보다 적은 갯수의 특수 목적 레지스터를 갖는다. 대부분의 프로세서는 다음과 같은 특수 목적의 전용 레지스터를 가지고 있다.

프로그램 카운터 (Program Counter, PC) 이 레지스터는 다음에 실행할 명령어의 주소를 가지고 있다. 이 값은 명령어를 가져올 때마다 자동으로 증가한다.

스택 포인터 (Stack Pointer, SP) 프로세서는 데이터를 임시로 저장할 수 있는 대규모의 외부 RAM에 접근해야 한다. 스택은 외부 메모리에 임시로 데이터를 저장하고 다시 읽어들일 수 있는 손쉬운 방법 중 하나이다. 대개 프로세서들은 스택에 데이터를 넣고 (push), 나중에 이를 다시 가져오는 (pop) 특별한 명령어들을 가지고 있다. 스택은 "마지막에 들어온 것이 맨 먼저 나가는 (last in first out, LIFO)" 방식으로 동작한다. 다르게 말하면, 스택에 두개의 값 x와 y를 집어넣고, 값을 빼내면 먼저 y값을 먼저 얻게 되는 것이다.

어떤 프로세서에서는 스택이 위로 자라지만, 다른 프로세서는 메모리가 시작하는 쪽인 아래쪽으로 스택이 자란다. ARM같은 프로세서는 두가지 방식 모두를 지원한다.

프로세서 상태 (Processor Status, PS) 어떤 명령어들은 실행하면 결과가 나오는 것이 있다. 예를 들어 "레지스터 X의 값이 Y의 값보다 큰가?"라는 명령을 수행하면 예 또는 아니오의 결과가 나온다. PS 레지스터는 이런 값과 함께, 프로세서의 현재 상태를 나타내는 다른 정보들을 가지고 있다. 이런 예로, 대부분의 프로세서는 커널 모드(또는 관리자 모드)와 사용자 모드라는 두가지 이상의 동작모드를 가지고 있는데, PS 레지스터는 현재 어떤 모드에 있는지 나타내는 있는 정보를 가지고 있다.

1.2 메모리(Memory)

모든 시스템에는 메모리 분류 체계가 있으며, 다른 크기와 속도를 갖는 메모리들이 이 체계의 서로 다른 지점에 위치한다. 우선 가장 빠른 메모리는 캐시 메모리로, 말 그대로 메인 메모리의 내용을 임시로 보관하는, 즉 캐시하는데 사용하는 메모리이다. 이런 메모리는 속도는 매우 빠르지만 값이 비싸기 때문에, 대부분의 프로세서는 칩 안에 소량의 캐시 메모리를, 그리고 보드상에 추가로 캐시 메모리를 가지고 있다. 어떤 프로세서는 하나의 캐시에서 명령어와 데이터를 같이 갖지만, 명령어와 데이터 용으로 두 개의 캐시를 갖는 것도 있다. 알파 AXP 프로세서는 두개의 내장 메모리 캐시를 가지고 있는데, 하나는 데이터용이고(D-캐시), 다른 하나는 명령어용이다(I-캐시). 외장 캐시(B-캐시)는 이 두가지를 함께 가진다. 마지막으로 외장 캐시 메모리에 비해 매우 느린 메인 메모리가 있다. CPU 칩상에 있는 캐시와 비교하면 메인 메모리는 정말 밥통같은 것이다

캐시와 메인 메모리는 같은 값을 유지하고 있어야 한다 (일치성). 다르게 말하면, 메인 메모리에 있는 어떤 데이터가 캐시의 하나 이상의 위치에 저장되어 있을 때, 시스템은 캐시에 있는 값과 메모리에 있는 값이 일치하도록 해주어야 한다는 것이다. 캐시의 일치성은 어떤 부분은 하드웨어에 의해, 어떤 부분은 운영체제에 의해 유지된다. 이런 것은 소기의 목적을 달성하기 위해 하드웨어와 소프트웨어가 밀접하게 협동해야 하는, 시스템의 다른 주요 작업들에 있어서도 마찬가지다.

1.3 버스(Bus)

시스템 보드상의 개개 구성요소들은 여러개의 버스라는 연결시스템으로 상호 연결되어 있다. 시스템 버스는 세가지 논리적인 기능 요소로 나누어지는데, 하나는 주소 버스(address bus), 다른 하나는 데이터 버스(data bus), 나머지 하나는 제어 버스(control bus)이다. 주소 버스는 데이터를 전송할 메모리의 위치(주소)를 지정한다. 데이터 버스는 전송되는 데이터를 가지고 있으며, 양방향으로 전송 가능하다. 즉 CPU로 읽어 들이거나 CPU에서 쓰는 것이 가능하다. 제어 버스는 시스템 전체에 타이밍 신호와 제어 신호를 전달하는 여러 선들을 가지고 있다. 여러 방식의 버스가 있지만, ISA나 PCI 버스가 주변장치를 시스템에 연결하는 대중적인 방법으로 사용되고 있다.

1.4 컨트롤러와 주변장치

주변장치는 시스템 보드 상이나 또는 보드에 꽂힌 카드에 있는 컨트롤러 칩에 의해 제어되는, 그래픽 카드나 디스크같이 실제로 존재하는 장치를 말한다. IDE 디스크는 IDE 컨트롤러 칩에 의해, SCSI 디스크는 SCSI디스크 컨트롤러 칩에 의해 제어된다. 이들 컨트롤러는 여러 종류의 버스를 통해, CPU와 다른 컨트롤러들과 서로 연결되어 있다. 요즘 나오는 시스템의 대부분은 이들 주요 시스템 구성요소들을 연결하기 위해 PCI와 ISA 버스를 사용한다. 컨트롤러는 CPU와 비슷한 하나의 프로세서이고, CPU 입장에서는 똑똑한 도우미이다. CPU는 시스템 전체를 제어하는 것이다.

모든 컨트롤러는 서로 다르지만, 자신을 제어하기 위한 레지스터를 가지고 있다는 점은 비슷하다. CPU에서 실행되는 소프트웨어는 이들 제어용 레지스터를 읽고 쓸 수 있어야 한다. 어떤 레지스터는 에러를 나타내는 상태를 가지고 있기도 하고, 또다른 레지스터는 컨트롤러의 모드를 바꾸는 것 같은 제어 용도로 사용되기도 한다. CPU는 버스상에 있는 컨트롤러 각각에 개별적으로 주소지정을 할 수 있다. 이리하여 소프트웨어 디바이스 드라이버가 컨트롤러를 제어하기 위해 레지스터를 쓸 수 있게 된다. IDE 리본이 좋은 예로, 이는 버스상에 있는 드라이브를 따로따로 접근할 수 있도록 해준다. 다른 좋은 예로는 각 디바이스(그래픽카드같은)들을 서로 독립적으로 접근할 수 있는 PCI 버스가 있다.

1.5 주소공간(Address Space)

CPU와 메인 메모리를 연결하는 시스템 버스는, CPU와 다른 하드웨어 주변장치를 연결하는 버스와는 분리되어 있다. 하드웨어 주변장치가 존재하고 있는 메모리 공간을 총괄하여 I/O 공간이라고 한다. I/O 공간은 더 쪼갤 수 있지만, 당분간 이에 대해 생각하지 않도록 하자. CPU는 시스템 공간 메모리와 I/O 공간 메모리에 모두 접근 가능하지만, 컨트롤러는 단지 시스템 메모리에 간접적으로 접근할 수 있을 뿐이며, 이것도 CPU의 도움을 받아야만 한다. 장치의 입장에서 보면, 가령 플로피 디스크 컨트롤러라고 한다면, 자신의 제어 레지스터가 있는 주소공간(ISA)만 보일 뿐, 시스템 메모리는 보이지 않을 것이다. 일반적으로 CPU는 메모리 공간과 I/O 공간을 접근하는데 다른 명령어를 사용한다. 예를 들어, "I/O 공간 0x3f0 주소에서 한 바이트를 읽어 레지스터 X에 저장하라"같은 명령이 있는 것이다. 이는 CPU가 I/O 공간에 있는 주변장치의 레지스터를 읽고 씀으로써, 하드웨어 주변장치를 제어하는 방법을 그대로 보여준다. 일반적으로 쓰이는 주변장치들(IDE 컨트롤러, 직렬포트, 플로피 디스크 컨트롤러 등)의 레지스터가 있는 I/O 공간은 PC 구조가 개발된 후 오랫동안 관례에 의해 고정되어 있다. I/O 공간의 주소 0x3f0은 직렬포트 COM1 제어 레지스터 중 하나의 주소이다.

가끔은 컨트롤러가 많은 양의 데이터를 시스템의 메모리에서 읽어 들이거나 메모리로 써 넣어야 할 경우가 있다. 사용자의 데이터를 하드디스크에 기록하는 경우가 이런 좋은 예이다. 이 때는, DMA (Direct Memory Access, 직접 메모리 접근) 컨트롤러를 사용하여 하드웨어 주변장치가 바로 시스템 메모리에 접근할 수 있게 한다. 하지만 이것 역시 CPU의 엄격한 제어와 감시하에 이루어진다.

1.6 타이머

모든 운영체제는 현재 시간을 알 필요가 있기 때문에, 지금 나오는 PC들은 RTC(Real Time Clock, 실시간 클럭)라는 특수한 주변장치를 가지고 있다. 이것은 정확한 시간과, 정밀한 시간 간격을 제공하는 두가지 역할을 한다. RTC 는 자체 배터리를 가지고 있어서, PC의 전원을 끄더라도 계속 동작한다. 이것이 PC가 항상 정확한 날짜와 시간을 알 수 있는 방법이다. 간격 타이머(interval timer)는 운영체제가 중요한 작업의 일정을 정확하게 조절할 수 있게 해준다.

번역 : 이호, 김진석, 이대현, 이준희, 고양우, truejaws
정리 : 이호

2장

소프트웨어의 기초

프로그램이란 특정한 작업을 수행하는 컴퓨터 명령어들의 집합이다. 프로그램은 어셈블리어와 같이 저급 컴퓨터 언어로 작성할 수도 있고, C 프로그래밍 언어처럼 기계와 무관한 고급 언어로 작성할 수도 있다. 운영체제는 사용자가 스프레드시트나 워드 프로세서와 같은 응용 프로그램을 실행할 수 있도록 해주는 특별한 프로그램이다. 이 장에서는 프로그래밍의 기본 원칙과 운영체제의 목표와 기능에 대한 개요를 제시하고자 한다.

2.1 컴퓨터 언어(Computer Language)

2.1.1 어셈블리어(Assembly Language)

CPU가 메모리에서 가져와 실행하는 명령어는 사람이 전혀 이해할 수 없는 것이다. 이들은 컴퓨터가 정확히 무엇을 해야할 지 말해주는 기계어 코드이다. 인텔 80486 CPU에서 십육진수 0x89E5는 ESP 레지스터의 내용을 EBP 레지스터로 복사하라는 명령이다. 초창기 컴퓨터를 위해 개발된 최초의 소프트웨어 도구 중 하나는 어셈블러였다. 어셈블러는 사람이 읽을 수 있는 형태의 소스 파일을 어셈블하여 기계어 코드를 만드는 프로그램이다. 어셈블리어는 레지스터와 자료에 대한 연산을 명시적으로 다루며, 마이크로프로세서마다 다르다. 인텔 x86 마이크로프로세서용 어셈블리어와 알파 AXP 마이크로프로세서용 어셈블리어는 완전히 다르다. 다음 알파 AXP용 어셈블리 코드는 프로그램이 수행할 수 있는 연산의 예를 보여준다.

```
      ldr r16, (r15)    ; Line 1
      ldr r17, 4(r15)   ; Line 2
      beq r16, r17, 100 ; Line 3
      str r17, (r15)    ; Line 4
100:                    ; Line 5
```

첫번째 문장(Line 1)은 레지스터15가 가진 주소에 있는 값을 레지스터16으로 읽어들인다. 그 다음 명령은 메모리 다음 위치의 내용을 레지스터17로 읽어들인다. 세 번째 줄에서는 레지스터16과 레지스터17의 내용을 비교하여, 이 값이 같으면 레이블100으로 분기한다. 두 레지스터에 들어있는 값이 같지 않다면, 프로그램은 네 번째 줄로 계속 진행하여 레지스터17의 내용을 메모리에 저장한다. 두 레지스터가 같은 값을 갖고 있다면, 그 값을 저장할 필요가 없다. 어셈블리 수준의 프로그램은 따분하고, 작성하는데 잔꾀가 많이 필요하며, 오류를 범하기 쉽다. 리눅스 커널 중에서 어셈블리어로 작성된 부분은 극히 일부에 지나지 않는다. 이들은 단시 효율성을 위해 어셈블리어로 작성되었으며, 특정 마이크로프로세서에 고유하다.

2.1.2 C 프로그래밍 언어와 컴파일러(Compiler)

어셈블리어로 큰 프로그램을 작성하는 것은 어려울 뿐만 아니라 시간도 많이 필요하다. 게다가 오류를 범하기 쉽고, 특정 프로세서에만 국한되므로 이식성도 없다. 그래서 C같이 기계에 무관한 언어를 사용하는 것이 훨씬 좋다. C는 프로그램을 처리할 논리적인 자료와 논리적인 알고리즘으로 표현할 수 있게 해준다. 컴파일러라고 하는 특수한 프로그램은 이 C 프로그램을 읽어서 어셈블리로 변환하여, 특정 기계에 해당하는 코드를 만들어낸다. 좋은 컴파일러는 훌륭한 어셈블리 프로그래머가 작성한 것에 가깝게 효율적인 어셈블리 코드를 만들어낸다. 리눅스 커널의 대부분은 C언어로 되어 있다. 다음 C 코드는 앞에 예로 든 어셈블리 코드와 똑같은 연산을 수행한다.

```
if (x != y)
      x = y;
```

이는 변수 x의 값과 변수 y의 값이 다르면 x에 y의 값을 복사할 것이다. C 코드는 각기 다른 일을 수행하는 여러개의 루틴들로 이루어진다. 루틴은 어떤 값이나, C언어에서 지원하는 자료형을 리턴할 수 있다. 리눅스 커널같이 큰 프로그램은 많은 수의 C 모듈로 이루어지며, 각 모듈은 자신만의 자료구조와 루틴들로 구성되어 있다. 이런 C 소스 코드 모듈이 모여서 파일 시스템을 다루는 것같은 논리적인 기능을 하게 된다.

C는 여러 가지 변수형을 지원한다. 변수란 심볼 이름으로 참조할 수 있는 메모리 상의 한 위치이다. 프로그래머는 이런 변수가 메모리 상의 어디에 있는지 신경 쓸 필요가 없다. 이 일은 밑에서 설명할 링커가 알아서 해준다. 변수는 각각 정수, 실수, 포인터 등의 다른 종류의 자료를 가질 수 있다.

포인터는 어떤 자료의 메모리 상의 위치인 주소를 값으로 가지는 변수이다. 어떤 변수 x가 메모리 상의 주소 *0x80010000*에 있다고 하자. 여기서 x를 가리키는 포인터 변수 - 이것을 px라고 하자 - 를 만들 수 있고, 이 px는 *0x80010030* 번지에 있다고 하자. 그러면 px의 값은 변수 x의 주소인 *0x80010000*이게 된다.

C에서는 서로 관련된 변수 여러개를 묶어 하나의 자료구조로 묶을 수 있다. 예를 들어,

```
struct {
      int i;
      char b;
} my_struct;
```

는 i라는 정수(32비트 자료공간을 차지한다)와, b라는 문자(8비트 자료), 이 두 개의 원소를 가진 my_struct라는 자료구조를 정의한다.

2.1.3 링커(Linker)

링커는 여러개의 오브젝트 모듈과 라이브러리를 연결하여 하나의 완결된 프로그램을 만들어내는 프로그램이다. 오브젝트 모듈은 어셈블러나 컴파일러가 만들어 낸 기계어 코드 출력물로, 기계어 코드와 자료, 그리고 링커가 다른 모듈과 결합하여 하나의 프로그램을 만들어 내는데 필요한 정보를 포함한다. 예를 들어 어떤 프로그램에서, 필요한 데이터베이스 함수를 모두 어떤 하나의 모듈이 가지고 있고, 명령행 인자를 처리하는 함수를 다른 모듈이 가지고 있다고 하자. 링커는 하나의 오브젝트 모듈에서 실제로 다른 모듈에 있는 자료구조나 루틴을 참조하고 있을 때, 이들 모듈 사이의 참조를 맞추어 준다. 리눅스 커널은 많은 요소의 오브젝트 모듈들을 링크하여 만든, 하나의 거대한 프로그램이다.

2.2 운영체제(Operating System)란 무엇인가?

소프트웨어가 없다면 컴퓨터는 그저 열이나 내는 전자제품 덩어리에 지나지 않는다. 하드웨어를 컴퓨터의 심장이라고 한다면, 소프트웨어는 컴퓨터의 영혼이라 할 수 있다. 운영체제는

사용자가 응용프로그램을 실행할 수 있도록 해주는 시스템 프로그램들을 모아놓은 것이다. 운영체제는 실제 하드웨어를 추상화하여 시스템의 사용자와 응용프로그램에게 가상 기계(virtual machine)를 제공한다. 그래서 실제로 운영체제가 시스템의 특성을 제공해주는 것처럼 느껴진다. 대부분의 PC는 하나 이상의 운영체제를 돌릴 수 있으며, 각 운영체제는 매우 다른 모습과 느낌을 갖고 있다. 리눅스는 운영체제를 구성하는 여러개의 기능적으로 분리된 조각들로 만들어진다. 리눅스에서 명백하게 구분되는 부분은 커널이지만, 라이브러리나 쉘이 없다면 커널은 무용지물이다.

운영체제가 무엇인지 이해를 할 수 있도록, 다음과 같이 간단한 명령을 쳤을 때 어떤 일이 나는지 생각해보자.

```
$ ls
Mail        c           images            perl
docs        tcl
$
```

여기서 $는 로그인 쉘(이 경우에는 bash)이 내보내는 프롬프트이다. 이는 사용자가 어떤 명령을 내리기를 기다리고 있다는 것을 의미한다. ls라고 쳐 넣으면 키보드 드라이버는 무슨 글자가 입력되었는지 인식하고 인식한 글자들을 쉘에 넘겨준다. 쉘은 그런 이름을 가진 실행 이미지가 있는지 찾고, 여기서는 /bin/ls라는 이미지를 찾게 된다. 커널 서비스를 호출하여 ls라는 실행 이미지를 가상 메모리에 올리고, 이를 실행하게 된다. ls 이미지는 커널의 파일 서브시스템의 함수를 호출하여 어떤 파일들이 있는지 찾는다. 파일 시스템은 캐시된 파일 시스템 정보를 이용하거나, 디스크 디바이스 드라이버를 사용하여 디스크에서 이 정보를 읽어올 수도 있다. 또는 파일 시스템이 네트웍 파일 시스템(Network File System, NFS)을 통하여 원격으로 마운트된 경우, 액세스해야 하는 원격 파일들의 세부정보를 찾기 위해 네트웍 드라이버를 이용하여 원격 기계와 정보를 교환할 수도 있다. 어떤 방법으로 정보를 찾았던 간에, ls는 그 정보를 출력하고, 비디오 드라이버는 이를 화면에 표시한다.

얘기가 좀 복잡해진 것 같지만, 어쨌든 이런 간단한 명령을 통해서도, 운영체제는 사실상 서로 협동하는 여러 기능들이 모여서 사용자에게 시스템의 일관된 모습을 보여준다는 것을 알 수 있다.

2.2.1 메모리 관리(Memory Management)

자원 - 예를 들어 메모리 - 이 무한히 있다면 운영체제가 하는 일의 상당 부분은 필요없는 일이 될 것이다. 모든 운영체제의 기본기 중의 하나는 적은 양의 실제 메모리(physical memory)를 많이 있는 것처럼 보이게 하는 것이다. 겉으로 보기에 많아 보이는 이 메모리를 가상 메모리(virtual memory)라고 부른다. 이 아이디어는 시스템 내에서 실행중인 소프트웨어를 속여서 메모리가 많이 있는 것처럼 믿게 만드는 것이다. 시스템은 메모리를 쉽게 다룰 수 있도록 페이지(page)로 쪼개고, 시스템이 실행되면서 이들 페이지를 하드디스크로 스왑(swap)한다. 소프트웨어는 멀티프로세싱이라는 또 다른 트릭 때문에 이 사실을 깨닫지 못한다.

2.2.2 프로세스(Process)

프로세스란 실행중인 프로그램이며, 각 프로세스는 각기 하나의 프로그램을 실행하는 구분된 개체이다. 현재 사용하는 리눅스 시스템에 동작하고 있는 프로세스를 살펴본다면, 상당히 많은 수의 프로세스가 있음을 알 수 있을 것이다. ps라고 타이핑하면 시스템에 있는 프로세스들을 보여주는데, 예를 들어 다음과 같은 결과가 나온다.

```
$ ps
  PID TTY STAT TIME COMMAND
  158 pRe 1    0:00 -bash
```

```
  174 pRe 1    0:00 sh /usr/X11R6/bin/startx
  175 pRe 1    0:00 xinit /usr/X11R6/lib/X11/xinit/xinitrc --
  178 pRe 1 N  0:00 bowman
  182 pRe 1 N  0:01 rxvt -geometry 120x35 -fg white -bg black
  184 pRe 1 <  0:00 xclock -bg grey -geometry -1500-1500 -padding 0
  185 pRe 1 <  0:00 xload -bg grey -geometry -0-0 -label xload
  187 pp6 1    9:26 /bin/bash
  202 pRe 1 N  0:00 rxvt -geometry 120x35 -fg white -bg black
  203 ppc 2    0:00 /bin/bash
 1796 pRe 1 N  0:00 rxvt -geometry 120x35 -fg white -bg black
 1797 v06 1    0:00 /bin/bash
 3056 pp6 3 <  0:02 emacs intro/introduction.tex
 3270 pp6 3    0:00 ps
$
```

만약 시스템에 CPU가 여러개 있다면 각 프로세스는 각기 다른 CPU에서 실행될 수 있을 것이다 (최소한 이론적으로는 그렇다). 하지만 불행히도 CPU는 보통 하나밖에 없기 때문에 운영체제는 각각의 프로세스를 돌아가며 짧은 시간 실행하는 또 다른 트릭을 사용해야 한다. 이 짧은 시간을 타임 슬라이스(time-slice)라고 한다. 이런 트릭을 멀티프로세싱(multi-processing) 또는 스케쥴링(scheduling)이라고 부르며, 이는 각 프로세스가 자신만이 유일한 프로세스인 것처럼 생각하도록 속이는 것이다. 프로세스 간에는 서로 보호가 되기 때문에 한 프로세스가 박살이 나거나 오동작을 해도 다른 프로세스에 영향을 미치지 않는다. 운영체제는 각 프로세스에게 자신만이 액세스 할 수 있는 분리된 주소공간을 줌으로써 이 기능을 달성한다.

2.2.3 디바이스 드라이버(Device Driver)

디바이스 드라이버는 리눅스 커널의 주요 부분을 구성한다. 디바이스 드라이버는 운영체제의 다른 부분들과 마찬가지로 특권층에서 동작하므로, 잘못될 경우 심각한 결과를 가져온다. 디바이스 드라이버는 자신이 제어하는 하드웨어 장치와 운영체제 간의 상호작용을 제어한다. 예를 들어, 파일 시스템은 IDE 디스크에 블럭을 기록할 때 일반적인 블럭 장치 인터페이스를 사용하는데, 디바이스 드라이버는 장치의 세세한 부분까지 챙기며, 장치마다 다른 일들을 실행한다. 디바이스 드라이버는 구동하려는 컨트롤러 칩에 따라 다르다. 그래서 NCR810 SCSI 컨트롤러가 있다면 NCR810 SCSI 드라이버가 필요한 것이다.

2.2.4 파일 시스템(File System)

유닉스와 마찬가지로, 리눅스에서도 시스템이 사용할 수 있는 구분된 파일 시스템에 접근하는데 장치 식별자(드라이브 번호나 드라이브 이름같은)를 사용하지 않는다. 대신 파일 시스템 전체를 하나의 계층적인 트리 구조로 연결하여 하나의 개체로 보여준다. 리눅스는 각각의 새로운 파일 시스템을 `/mnt/cdrom`같은 마운트 디렉토리에 마운트하여, 하나의 파일 시스템 트리 구조에 추가한다. 예를 들면 CD-ROM을 `/mn/cdrom`으로 마운트하는 것이다. 여러 가지 파일 시스템을 지원하는 것은 리눅스의 가장 중요한 특징 중의 하나이다. 이는 리눅스를 매우 유연하게 만들며, 다른 운영체제와 잘 공존할 수 있게 한다. 리눅스에서 가장 많이 사용하는 파일 시스템은 EXT2 파일시스템으로, 대부분의 리눅스 배포판이 EXT2를 지원한다.

파일 시스템에 의해 사용자는 파일 시스템의 형태나 그 하부의 물리적인 장치의 특징에 상관없이 시스템의 하드 디스크에 있는 파일이나 디렉토리를 인식할 수 있게 된다. 리눅스는 MS-DOS나 EXT2 등의 많은 다른 파일 시스템을 투명하게 지원하며, 마운트되어 있는 모든 파일과 파일 시스템을 하나의 통합된 가상 파일 시스템(Virtual File System, VFS)으로 제공한다. 따라서, 사용자와 프로세스는 일반적으로 어떤 파일이 무슨 파일 시스템에 속해 있는지 알 필요 없이 사용하기만 하면 된다.

블럭 디바이스 드라이버는 실제 블럭 장치의 유형(IDE와 SCSI같은)에 따른 차이점을 숨겨주기 때문에, 파일 시스템에 있어서는 이 물리적 장치는 그저 연속된 데이터 블럭의 모음일 뿐이다. 블럭의 크기는 장치에 따라 다를 수 있다. 예를 들어, 플로피 장치는 공통적으로 512바이트를 사용하는데 반해, IDE 장치는 1024바이트를 사용한다. 이 차이는 시스템 사용자에겐 보이지 않는다. EXT2 파일 시스템이 어떤 장치에 들어있든 간에 사용자에겐 모두 똑같이 보인다.

2.3 커널 자료구조(Kernel Data Structure)

운영체제는 시스템의 현재 상태에 대한 매우 많은 양의 정보를 갖고 있어야 한다. 시스템 내부에서 어떤 일이 일어나면 현재 상태를 반영하기 위해 이들 자료구조를 변경해야 한다. 예를 들어, 한 사용자가 시스템에 로그인하면 새로운 프로세스가 만들어지게 되는데, 커널은 이 새로운 프로세스를 나타내는 자료구조를 만들고, 이를 시스템 내의 다른 프로세스를 나타내는 모든 자료구조와 연결하여야 한다.

이들 자료구조의 대부분은 실제 메모리 상에 존재하는 것이며, 커널과 커널의 서브시스템만이 액세스할 수 있다. 자료구조는 데이터와 포인터를 포함하며, 이 포인터는 다른 자료구조나 루틴을 가리킨다. 리눅스 커널이 사용하는 자료구조를 한번에 뭉쳐서 보면 매우 혼동스러울 수도 있다. 모든 자료구조는 고유의 목적을 갖고 있으며, 일부는 여러 커널 서브시스템에서 사용하지만, 실제로는 처음 보기보다는 더 단순하다.

리눅스 커널을 이해하는 것은 리눅스 커널의 자료구조와 커널에 있는 여러 함수들이 이를 어떻게 활용하는지 이해하는데 달려 있다. 이 책은 리눅스 커널을 자료구조에 기반하여 설명한다. 각 커널 서브시스템을 원하는 일을 어떻게 처리하는지를 나타내는 알고리즘과, 커널의 자료구조를 어떻게 사용하는지를 중심으로 설명한다.

2.3.1 연결 리스트(Linked List)

리눅스는 자료구조를 서로 연결하기 위하여 여러 가지 소프트웨어 공학적 기법을 사용한다. 많은 경우 리눅스는 연결된(linked), 또는 연쇄된(chained) 자료구조를 사용하고 있다. 각 자료구조가 어떤 것 - 예를 들어 프로세스나 네트웍 장치 - 의 한 존재나 경우를 나타낸다면, 커널은 이들 모두를 찾아낼 수 있어야 한다. 연결 리스트에서는 루트 포인터가 리스트에 있는 첫 번째 자료구조(또는 원소)의 주소를 가지고, 각 자료구조는 리스트의 다음 원소의 주소를 가진다. 마지막 원소의 다음 원소를 가리키는 포인터는 리스트의 끝임을 나타내기 위해 0 또는 NULL 값을 가진다. 이중 연결 리스트(Doubly Linked List)에서는 각 원소가 다음 원소를 가리키는 포인터와 함께, 이전 원소를 가리키는 포인터도 가진다. 이중 연결 리스트를 사용하면 메모리 액세스 횟수가 더 많아지긴 하지만, 리스트의 중간에 원소를 추가하거나 삭제하는 것이 더 쉽다. 이는 운영체제에서 가장 전형적인 트레이드 오프(trade off)[7]이다. 메모리 액세스를 더 할 것인가, 아니면 CPU 사이클을 더 쓸 것인가.

2.3.2 해시 테이블(Hash Table)

연결 리스트는 자료구조를 묶는 손쉬운 방법이지만, 연결 리스트를 탐색하는 것은 비효율적일 수 있다. 어떤 특정 원소를 찾으려고 할 때, 원하는 걸 발견할 때까지 리스트 전체를 쭉 훑어보아야 하기 때문이다. 이런 제한을 피하기 위해 리눅스는 해싱(hashing)이라는 기법을 사용한다. 해시 테이블은 포인터의 배열, 즉 포인터의 벡터(vector)이다. 배열, 즉 벡터는 어

역주 7) 메모리를 더 액세스하고 CPU 사이클을 적게 쓸 것인가, 또는 메모리 액세스를 적게하고 CPU 사이클을 더 쓸 것인가 하는 갈림길에서 둘 사이의 타협점을 찾는 것을 말한다. (flyduck)

떤 것들이 메모리 상에 하나씩 이어져 있는 것을 말한다[8]. 즉 책꽂이는 책의 배열이라고 할 수 있다. 배열은 배열에서의 위치를 나타내는 인덱스(index)를 가지고 액세스한다. 책꽂이 비유를 조금 더 확장한다면, 각각의 책을 '다섯번째 책'과 같은 방식으로 책꽂이에서의 위치로 표현하는 것이다.

해시 테이블은 자료구조에 대한 포인터의 배열이며, 인덱스는 자료구조의 내용으로부터 만들어진다. 어떤 마을의 인구 분포를 나타내는 자료구조가 있다면, 이를 표현하는데 사람의 나이를 인덱스 값으로 쓸 수 있을 것이다. 이 경우 어떤 사람의 자료를 찾으려고 한다면 그 사람의 나이를 인덱스로 하여 인구 해시 테이블로부터 포인터를 얻고, 그 포인터를 따라가면 그 사람의 상세자료가 들어있는 자료구조가 나올 것이다. 불행히도 마을에는 같은 나이를 가진 사람이 많이 있을 수 있다. 그런 경우에는 그 포인터가 같은 나이를 가진 사람들의 연결 리스트를 가리키는 포인터가 된다. 물론 이 짧은 리스트를 찾는 것이 자료구조 전체를 뒤지는 것보다는 여전히 빠를 것이다.

해시 테이블은 자주 사용하는 자료구조로의 액세스 속도를 높여주기 때문에, 리눅스는 캐시를 구현하기 위해 해시 테이블을 종종 사용한다. 캐시는 빨리 액세스되어야 하는 바로 쓸 수 있는 정보이며, 대개 참조할 수 모든 정보의 일부만을 가지고 있다. 자료구조를 캐시에 넣어두는 것은 커널이 그 자료구조를 자주 액세스하기 때문이다. 캐시는 간단한 연결 리스트나 해시 테이블에 비하여 사용하고 관리하기가 복잡하다는 단점이 있다. 찾으려는 자료구조가 캐시에 있다면 (이를 캐시 히트라고 부른다) 아주 좋은 일이다. 그러나 만약 캐시에 없으면 관련된 자료구조를 모두 뒤져야 하고, 원하는 자료구조가 실제로 있으면 그것을 캐시에 추가하여야 한다. 새로운 자료구조를 캐시에 넣으려면 옛날 것은 버려야 할 수도 있다. 리눅스는 어떤 것을 버려야 할 지 정해야 하는데, 이번에 버린 자료가 바로 다음에 필요한 것이 되는 위험도 있다.

2.3.3 추상 인터페이스(Abstract Interface)

리눅스는 종종 자신의 인터페이스를 추상화한다. 인터페이스란 특정 방법으로 동작하는 루틴과 자료구조의 모음이다. 예를 들어, 모든 네트웍 디바이스 드라이버는 특정한 자료구조를 이용하여 정해진 루틴들을 제공해야 한다. 이런 방법으로 장치마다 다른 코드로 된 아래 계층에서 제공하는 서비스(또는 인터페이스)를 사용하는 일반적인 코드 계층이 있게 된다. 네트웍 계층은 일반화 되어있고, 장치마다 고유한 코드는 표준 인터페이스를 제공하여 이를 지원한다.

종종 이들 하위 계층은 부팅할 때 상위 계층에 자신을 등록한다. 이러한 등록 과정은 대개 어떤 연결 리스트에 자료구조를 추가하는 일을 수반한다. 예를 들어, 커널에 들어 있는 각각의 파일 시스템은 부팅할 때 자신을 커널에 등록하며, 모듈을 사용하는 경우에는 처음으로 그 파일 시스템이 사용될 때 등록된다. 어떤 파일 시스템이 등록되어 있는 지를 보려면 `/proc/filesystems`를 들여다보면 된다. 때로 등록된 자료구조가 함수에 대한 포인터를 가지고 있는 경우도 있다. 이들 포인터는 특정한 업무를 수행하는 소프트웨어 함수의 주소이다. 다시 파일 시스템 등록을 예로 들어보면, 각 파일 시스템이 등록할 때 리눅스 커널에 넘겨주는 자료구조에는, 파일 시스템이 마운트될 때마다 불리는 파일 시스템에 고유한 루틴의 주소가 들어있다.

번역 : 고양우, 신문석,
정리 : 이호

역주 8) 이후에 벡터라는 용어는 배열과 같은 의미로 쓰인다. (flyduck)

3장

메모리 관리 (Memory Management)

메모리 관리 서브시스템은 운영체제에서 가장 중요한 부분 중 하나이다. 초창기의 컴퓨터에서부터, 시스템에 물리적으로 존재하는 것보다 더 많은 양의 메모리를 필요해왔다. 물리적인 메모리의 한계를 극복하기 위한 여러 기법들이 개발되었는데, 가상 메모리 기법이 가장 성공적이다. 가상 메모리(virtual memory)는 메모리를 필요로 하는 서로 경쟁하는 프로세스 사이에 메모리를 공유하도록 하여, 시스템이 실제 가진 것보다 더 많은 메모리를 가진 것처럼 보이도록 한다.

가상 메모리는 컴퓨터의 메모리를 늘리는 일만 하는 것은 아니다. 메모리 관리 서브시스템은 다음과 같은 것을 제공한다.

넓은 주소공간 운영체제는 시스템이 실제 가진 것보다 훨씬 많은 양의 메모리를 가지고 있는 것처럼 보이게 한다. 가상 메모리는 시스템의 물리적 메모리보다 몇 배나 더 클 수 있다.

보호 시스템의 각 프로세스는 각자의 독립된 가상 주소공간을 갖는다. 이들 가상 주소공간은 서로 완벽하게 분리되어 있어서, 어떤 응용프로그램을 실행하는 프로세스는 다른 것에 영향을 줄 수 없다. 또 하드웨어 가상 메모리 메커니즘은 메모리 영역에 쓰기를 금지할 수 있게 한다. 이것은 코드와 데이터가 나쁜 프로그램에 의해 덮어 쓰여지는 것을 막아준다.

메모리 매핑 메모리 매핑은 이미지와 데이터 파일을 프로세스의 주소공간에 매핑하기 위해 사용된다. 메모리 매핑에서 파일의 내용은 프로세스의 가상 주소공간에 직접 연결된다.

공정한 물리적 메모리 할당 메모리 관리 서브시스템은 시스템에서 실행중인 프로세스들이 서로 공정하게 물리적 메모리를 공유할 수 있게 한다.

공유 가상 메모리 가상 메모리는 프로세스들이 분리된 (가상) 주소공간을 가질 수 있도록 하지만, 때로는 프로세스들이 메모리를 공유하는 것이 필요할 때가 있다. 예를 들어 시스템에 명령쉘 bash를 실행하고 있는 여러 개의 프로세스가 있다고 하자. 각 프로세스의 가장 주소공간에 bash의 여러 복사본을 갖는 대신에, 물리적 공간에 하나의 복사본을 갖고 bash를 실행하는 모든 프로세스가 그것을 공유하는 것이 바람직하다. 동적 라이브러리는 여러 프로세스가 실행 코드를 공유하는 대표적인 예이다.

공유 메모리는 또한 두 개 이상의 프로세스가 그들 모두에게 공통적인 메모리를 통해 정보를 교환함으로써, 프로세스간 통신(IPC) 메커니즘으로 사용될 수 있다. 리눅스는 유닉스 시스템 V의 공유 메모리 IPC를 지원한다.

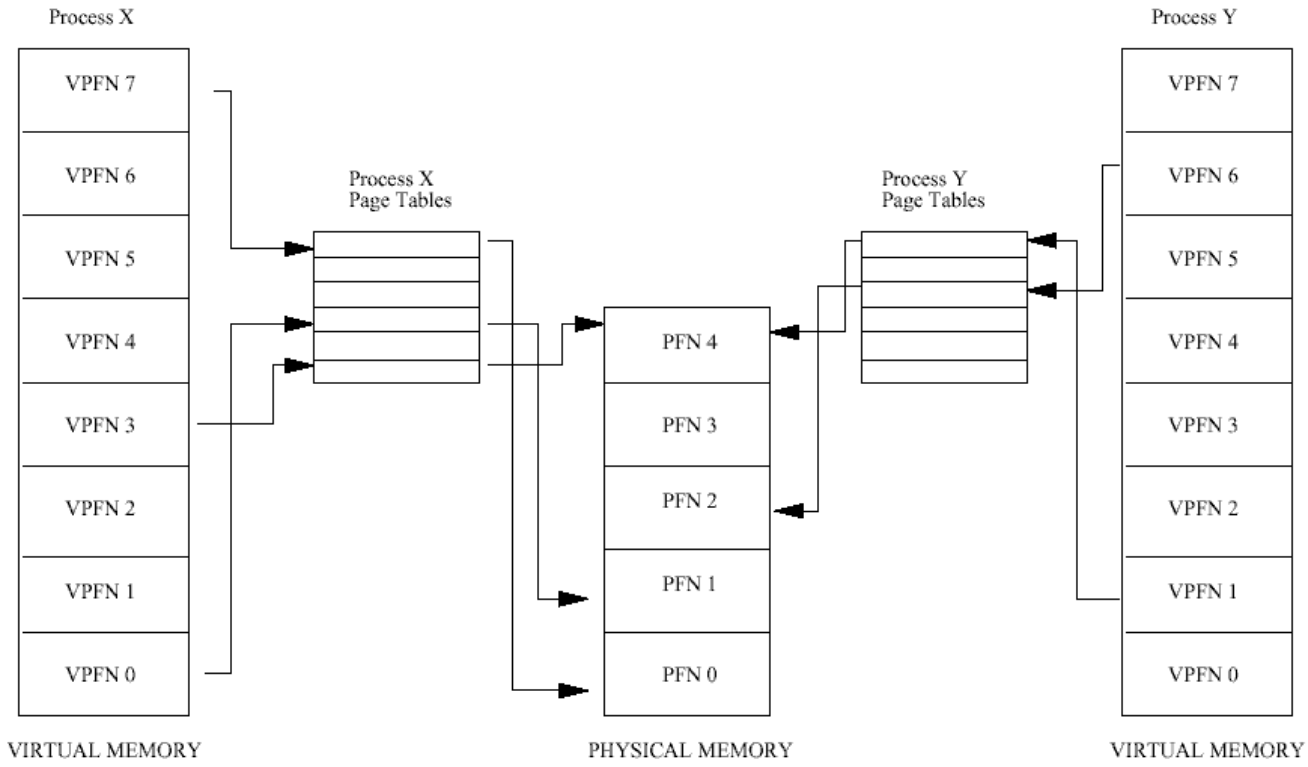


그림 3.1 : 가상 주소에서 물리 주소로 매핑의 추상적 모델

3.1 가상 메모리의 추상적 모델(abstract model)

리눅스가 가상 메모리를 지원하기 위해 사용하는 기법을 살펴보기 전에, 너무 자세히 파고들어 혼란스럽지 않도록 먼저 추상적 모델을 검토하는 것이 도움이 될 것이다.

프로세서가 프로그램을 실행할 때, 프로세서는 메모리로부터 명령어를 읽어 와서 해석한다. 명령을 해석하는 데에는 메모리의 어떤 위치에 있는 내용을 가져오거나 저장해야 하기도 한다. 프로세서는 명령어를 실행하고 프로그램의 다음 명령어로 이동한다. 이렇게 하여 프로세서는 언제나 명령어를 가져오거나, 데이터를 가져오거나 저장하기 위해 메모리에 접근한다.

가상 메모리 시스템에서 이 주소들은 모두 물리적 주소가 아니라 가상 주소이다. 이 가상 주소들은 운영체제가 관리하는 테이블들에 저장된 정보를 바탕으로 프로세서에 의해 물리적 주소로 변환된다.

이 변환을 쉽게 하기 위해 가상 메모리와 물리적 메모리는 페이지라는 작은 조각으로 나뉜다. 이 페이지들은 모두 같은 크기인데, 꼭 같은 크기일 필요는 없지만, 그렇지 않다면 시스템을 관리하기가 무척 어려워질 것이다. 리눅스는 알파 AXP 시스템에서는 8KB 페이지를, 인텔 x86 시스템에서는 4KB 페이지를 사용한다[9]. 각 페이지에는 페이지 프레임 번호(page frame number, PFN)라는 유일한 번호가 부여된다. 이와 같은 페이지 모델에서 가상 주소는 가상 페이지 프레임 번호와 오프셋, 두 부분으로 이루어진다. 페이지 크기가 4KB라면 가상 주소의 0비트에서 11비트는 오프셋을 나타내고, 12번 비트 이상은 가상 페이지 프레임 번호를 나타낸다[10]. 프로세서가 가상 주소를 처리할 때마다 오프셋과 가상 페이지 프레임 번호를

역주 9) 실제 인텔 80386에서 메모리를 4KB 페이지 단위로 다루고 있으며, 이 페이지 크기는 하드웨어에서 지원하는 크기를 따른 것이다. (flyduck)

역주 10) 4KB는 2^{12}이므로 이 한페이지의 주소를 나타내는데 12비트가 필요하다. 인텔 80385 CPU에서는 페이지 프레임 번호에 20비트, 오프셋에 12비트를 사용하여 모두 32비트의

추출해야 한다. 프로세서는 가상 페이지 프레임 번호를 물리적 페이지 프레임 번호로 변환하고 해당 물리적 페이지에서 오프셋에 해당하는 주소를 접근한다. 이렇게 하기 위해 프로세서는 페이지 테이블(page table)을 사용한다.

그림 3.1은 프로세스 X와 프로세스 Y 두 프로세스의 가상 주소공간과, 각자의 페이지 테이블을 보여준다. 이 페이지 테이블은 각 프로세스의 가상 페이지를 메모리의 물리적 페이지로 대응시킨다. 이 그림에서 프로세스 X의 가상 페이지 프레임 번호 0은 물리적 페이지 프레임 번호 1로 대응되고, 프로세스 Y의 가상 페이지 프레임 번호 1은 물리적 페이지 프레임 번호 4로 대응된다. 이론적으로 페이지 테이블은 다음과 같은 정보를 가진다 :

- 유효 플래그. 이것은 페이지 테이블 엔트리가 유효한가를 나타낸다.
- 이 엔트리가 기술하는 물리적 페이지 프레임 번호.
- 접근 제어(access control) 정보. 이것은 페이지가 어떻게 사용될 수 있는지 기술한다. 데이터를 기록할 수 있는가? 실행가능 한 코드를 포함하는가?

페이지 테이블은 가상 페이지 프레임 번호를 오프셋으로 사용하여 접근한다. 가상 페이지 프레임 5는 테이블의 6번째 항목이 된다 (0이 첫번째 항목이다)

가상 주소를 물리적 주소로 변환하기 위해, 프로세서는 먼저 가상 주소 페이지 프레임 번호와, 가상 페이지 안에서의 오프셋을 구해야 한다. 페이지 크기를 2의 제곱수로 하면, 이 계산은 비트마스크와 쉬프트 연산으로 쉽게 처리할 수 있다. 다시 그림 3.1에서, 페이지 크기가 *0x2000*바이트(8KB, 십진수로 8192)라면, 프로세서는 프로세스 Y의 가상 주소공간에서의 주소 *0x2194*를 가상 페이지 프레임 번호 1과 오프셋 *0x194*로 변환한다.

프로세서는 가상 페이지 프레임 번호를 인덱스로 프로세스의 페이지 테이블을 참조하여, 페이지 테이블 엔트리(page table entry, PTE)를 가져온다. 이 페이지 테이블 엔트리가 유효하다면, 프로세서는 이 엔트리에서 물리적 페이지 프레임 번호를 가져온다. 엔트리가 유효하지 않다면, 프로세스는 가상 메모리 공간에 존재하지 않는 영역을 접근한 것이다. 이 경우에 프로세서는 주소를 결정할 수 없고 운영체제에 제어를 넘겨서 운영체제가 처리하도록 한다.

프로세서가 운영체제에게, 정확하게 어떤 프로세스가 유효한 변환을 할 수 없는 가상 주소에 접근하려 했는지를 알리는 방법은 프로세서마다 다르다. 이것은 페이지 폴트(page fault)라고 하며, 프로세서가 이를 어떻게 전달하든지 간에, 운영체제는 폴트가 발생한 가상 주소와 페이지 폴트의 원인을 통보받는다.

그 페이지 테이블 엔트리가 유효한 경우, 프로세서는 물리적 페이지 프레임 번호에 페이지 크기를 곱해서 물리적 메모리에서의 베이스 주소를 얻는다. 마지막으로 프로세서는 오프셋을 더하여 필요한 명령이나 데이터에 도달한다[11].

위의 예를 다시 보면, 프로세스 Y의 가상 페이지 프레임 번호 1은 물리적 페이지 프레임 번호 4에 대응되고, *0x8000*(4 x *0x2000*)에서 시작된다. 여기에 0x194 바이트의 오프셋을 더하면 최종적인 물리적 주소 *0x8194*를 얻을 수 있다.

이렇게 가상 주소를 물리적 주소로 대응시킴으로써, 가상 메모리는 시스템의 물리적 페이지에 임의의 순서로 배열될 수 있다. 예를 들어, 그림 3.1의 프로세스 X의 가상 페이지 프레임 번호 0은 물리적 페이지 프레임 번호 1로 대응되는 반면, 가상 페이지 프레임 번호 7은 가상 페이지 프레임 번호 0보다 높음에도 물리적 페이지 프레임 번호 0으로 대응된다. 이것은 가상 메모리의 재미있는 부산물을 보여준다. 가상 메모리의 페이지들은 물리적 메모리에 어떤 특정한 순서로 존재하지 않아도 된다.

3.1.1 요구 페이징(Demand Paging)

주소공간 즉 4GB의 주소공간을 갖는다. (flyduck)

역주 11) 즉 물리적 주소는 Physical PFN * PAGE_SIZE + offset이다. (flyduck)

실제로 가상 메모리보다 훨씬 적은 물리적 메모리만 있기 때문에, 운영체제는 물리적 메모리가 비효율적으로 사용되지 않도록 주의해야 한다. 물리적 메모리를 절약하는 방법 하나는, 실행중인 프로그램이 현재 사용하는 가상 페이지만을 로드하는 것이다. 예를 들어, 데이터베이스 프로그램이 데이터베이스에 질의를 한다고 하자. 이 경우 모든 데이터베이스가 메모리에 로드될 필요는 없다. 검색할 데이터 레코드들만 있으면 된다. 데이터베이스 질의가 검색 질의라면, 데이터베이스 프로그램에서 새로운 레코드를 추가하는 것을 처리하는 부분의 코드를 읽어들일 필요는 없을 것이다. 이렇게 가상 페이지들이 접근되는 경우에만 메모리에 읽어들이는 기법을 요구 페이징이라고 한다.

프로세스가 현재 메모리에 없는 가상 주소를 접근하려고 하면, 프로세서는 참조된 가상 페이지에 대한 페이지 테이블 엔트리를 찾을 수 없을 것이다. 예를 들어, 그림 3.1에서 프로세스 X의 페이지 테이블에는 가상 페이지 프레임 번호 2에 대한 엔트리가 없으므로, 프로세스 X가 가상 페이지 프레임 번호 2에 포함된 주소에서 읽으려고 하면, 프로세서는 그 주소를 물리적 주소로 변환할 수 없을 것이다. 이 시점에서 프로세서는 운영체제에게 페이지 폴트가 발생했다고 통보한다.

만약 폴트가 발생한 가상 주소가 유효하지 않은 것이라면, 그 프로세스는 접근할 수 없는 가상 주소에 접근하려고 한 것이다. 대체로 이건 메모리의 아무 주소에나 값을 쓰는 것처럼, 응용프로그램이 잘못된 것이다. 이 경우 운영체제는 이 프로세스를 종료시켜, 시스템의 다른 프로세스들을 이 잘못된 프로세스로부터 보호한다.

만약 폴트가 발생한 가상 주소가 유효한 것인데, 주소가 가리키는 페이지가 메모리에 현재 없다면, 운영체제는 해당하는 페이지를 디스크의 이미지로부터 메모리에 가져와야 한다. 디스크 접근은 상대적으로 긴 시간이 걸리므로, 프로세스는 페이지가 도착할 때까지 한참을 기다려야 한다. 시스템에 실행할 수 있는 다른 프로세스가 있다면 운영체제는 이들 중 하나를 선택하여 실행한다. 가져온 페이지는 빈 물리적 페이지 프레임에 기록되고, 가상 페이지 프레임 번호를 위한 엔트리가 프로세스의 페이지 테이블에 추가된다. 이제 프로세스는 메모리 폴트가 발생했던 기계어 명령에서부터 재실행된다. 이번에 다시 가상 메모리 접근이 이루어질 때, 프로세서는 가상 주소를 물리적 주소로 변환할 수 있게 되고, 프로세스는 계속 실행된다.

리눅스는 실행 이미지를 프로세스의 가상 메모리에 로드하기 위해 요구 페이징을 사용한다. 명령을 실행할 때마다, 명령을 포함하는 파일을 열고, 파일의 내용이 프로세스의 가상 메모리로 매핑된다. 이것은 이 프로세스의 메모리 맵을 기술하는 자료구조를 변경하여 이루어지며, 이를 메모리 매핑이라고 한다. 어쨌든 이미지의 첫번째 부분만 실제로 물리적 메모리에 가져오며, 나머지 부분은 디스크에 남아 있다. 이미지가 실행됨에 따라 페이지 폴트가 발생하고, 리눅스는 프로세스의 메모리 맵을 사용하여 이미지의 어느 부분을 실행할 수 있도록 메모리에 가져올 지 결정한다.

3.1.2 스와핑(Swapping)

프로세스가 가상 페이지를 물리적 메모리에 가져와야 하는데, 비어 있는 물리적 페이지가 없다면, 운영체제는 물리적 메모리에서 다른 페이지를 제거하여, 가져올 페이지를 위해 공간을 마련해야 한다.

물리적 메모리에서 제거될 페이지가 이미지나 데이터 파일에서 온 것이고, 이 페이지에 쓰여진 것이 없다면, 페이지의 내용을 저장할 필요는 없다. 대신 그냥 제거를 하고, 나중에 다시 필요하게 되면 이미지나 데이터 파일로부터 다시 메모리에 읽어들이면 된다.

그러나 페이지가 변경되었다면, 운영체제는 페이지의 내용을 나중에 다시 사용할 수 있도록 보존해야 한다. 이런 페이지를 더티 페이지(dirty page)라고 하며, 이를 메모리에서 제거할 때 스왑 파일(swap file)이라는 특별한 파일에 저장한다. 스왑 파일에 접근하는 것은 프로세서나

물리적 메모리의 속도에 비해 매우 오래 걸리므로, 운영체제는 페이지를 디스크에 기록할 필요성과, 다시 사용될 수 있도록 메모리로 가져오게 될 필요성을 잘 다루어야 한다.

어떤 페이지를 제거 또는 스왑할지를 결정하기 위해 사용하는 알고리즘(스왑 알고리즘)이 효율적이지 않으면 쓰레싱(thrashing)[12]이라고 불리는 상태가 발생한다. 이 때 페이지는 계속 디스크에 기록되고 또 다시 읽어오게 되며, 운영체제는 너무 바빠서 실제 작업은 거의 못하게 된다. 예를 들어 그림 3.1에서, 물리적 페이지 프레임 번호 1이 계속 접근된다면, 이것은 하드디스크로 스와핑할 좋은 후보가 아니다. 프로세스가 현재 사용하고 있는 페이지의 집합을 작업 집합(working set)이라고 하는데, 효율적인 스왑 정책은 모든 프로세스들의 작업 집합이 모두 물리적 메모리에 있도록 한다.

리눅스는 시스템에서 제거될 페이지를 공정하게 선택하기 위해, 가장 최근에 사용된(Least Recently Used, LRU) 페이지 수명(page aging) 기법을 사용한다. 이 기법에서 시스템의 모든 페이지는, 그 페이지에 접근될 때마다 변경되는 수명을 갖고 있다. 페이지는 자주 접근될수록 젊어지고, 적게 접근될수록 나이가 들게 된다. 나이든 페이지는 스와핑의 좋은 후보이다.

3.1.3 공유 가상 메모리(Shared Virtual Memory)

가상 메모리는 여러 프로세스가 메모리를 쉽게 공유하게 해준다. 모든 메모리 접근은 페이지 테이블을 통해서 이루어지며, 각 프로세스는 독립된 페이지 테이블을 갖고 있다. 두 개의 프로세스가 물리적 메모리의 페이지를 공유하려면, 그 물리적 페이지의 프레임 번호가 두 프로세스의 페이지 테이블 모두에 페이지 테이블 엔트리로 있어야 한다.

그림 3.1은 두 프로세스가 물리적 페이지 프레임 번호 4를 공유하는 것을 보여준다. 이 물리적 페이지는 프로세스 X 입장에서 가상 페이지 프레임 번호 4이고, 프로세스 Y 입장에서 가상 페이지 프레임 번호 6이다. 이것은 페이지 공유의 재미있는 점을 보여준다. 공유되는 물리적 페이지는 이 물리적 페이지를 공유하는 어떤 프로세스에서도 가상 메모리의 같은 위치에 있을 필요가 없다.

3.1.4 물리적 주소 모드(Physical Addressing Mode)와 가상 주소 모드(Virtual Addressing Mode)

운영체제 자신이 가상 메모리에서 동작하는 것은 별 의미가 없다. 그렇게 되면 운영체제가 자신을 위해 페이지 테이블을 유지해야 하는 끔찍한 상황이 벌어질 것이다. 대부분의 범용 프로세서들은 물리적 주소 모드와 가상 주소 모드를 함께 제공한다. 물리적 주소 모드에서는 페이지 테이블이 필요없으며, 이 모드에서 프로세서는 아무런 주소 변환도 하지 않는다. 리눅스 커널은 물리적 주소공간에서 실행되도록 링크되어 있다.

알파 AXP 프로세서는 특별한 물리적 주소 모드를 갖고 있지 않다. 대신에 메모리 공간을 여러 부분으로 나누어, 그 중의 두 개를 물리적으로 매핑된 주소로 지정해 둔다. 이 커널 주소공간은 KSEG 주소공간이라고 부르며, *0xfffffc0000000000*부터 위쪽 주소 전부를 포함한다. KSEG에 링크된 코드(정의에 따라 커널 코드이다)를 실행하거나 KSEG의 데이터를 접근하기 위해서는 코드는 반드시 커널 모드에서 실행되어야 한다. 알파에서의 리눅스 커널은 주소 *0xfffffc0000310000*로부터 실행되도록 링크되어 있다.

3.1.5 접근 제어(Access Control)

페이지 테이블 엔트리는 접근 제어 정보도 가지고 있다. 프로세서는 프로세스의 가상 주소

역주 12) 계속 스왑 파일을 접근하느라 디스크만 고생하고 실제 작업은 실행되지 않는 현상 (심마로)

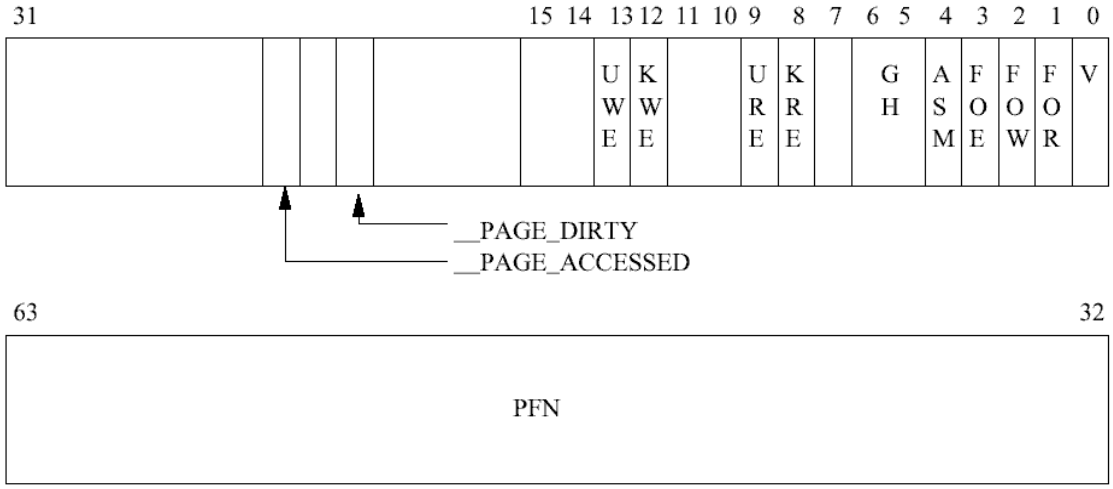


그림 3.2 : 알파 AXP 페이지 테이블 엔트리

를 물리적 주소로 변환하기 위해 이미 페이지 테이블 엔트리를 사용하기 때문에, 쉽게 접근 제어 정보를 사용하여, 이 프로세스가 허용되지 않은 방식으로 메모리를 접근하지 않도록 할 수 있다.

메모리 영역에 대한 접근을 제한하려고 하는 이유는 몇 가지가 있다. 실행 코드를 담고 있는 곳 같은 어떤 메모리는 자연히 읽기 전용 메모리이며, 운영체제는 프로세스가 자신의 실행 코드 위에 데이터를 쓰는 것을 허락해서는 안 된다. 반대로, 데이터를 담고 있는 페이지는 쓰여질 수 있지만 그 메모리를 명령어로 간주하여 실행하려는 시도는 실패해야 한다. 대부분의 프로세서는 적어도 두 가지 실행 모드 - 커널모드와 사용자모드 - 를 가지고 있다. 프로세서가 커널 모드로 수행중이 아니라면, 사용자가 커널 코드를 실행하거나 커널 자료구조에 접근하는 것을 막고 싶을 것이다.

접근 제어 정보는 PTE에 들어있으며 프로세서마다 다르다. 그림 3.2는 알파 AXP 프로세서의 PTE를 보여준다. 각 비트 필드의 의미는 다음과 같다 :

V 유효(Valid) 이 페이지 테이블 엔트리는 유효함.
FOE "실행시 오류(Fault on Execute)" 이 페이지의 명령을 실행하려고 할 때마다 프로세서는 페이지 오류를 발생하고 컨트롤을 운영체제에게 넘긴다.
FOW "쓰기시 오류(Fault on Write)" 위와 같으나 실행대신 이 페이지로 쓰려고 할 때 오류가 발생한다.
FOR "읽기시 오류(Fault on Read)" 위와 같으나 이 페이지에서 읽으려 할 때 오류가 발생한다.
ASM 주소공간 매치(Address Space Match). 변환 버퍼에서 일부 엔트리만을 지우려고 할 때 사용된다.
KRE 커널 모드에서 실행 중인 코드에서 이 페이지를 읽을 수 있음.
URE 사용자 모드에서 실행중인 코드에서 이 페이지를 읽을 수 있음.
GH 입도 힌트(granularity hint)는 블럭 전체을 여러개의 변환 버퍼 엔트리가 아닌 하나의 엔트리에 매핑할 때 사용된다.
KWE 커널 모드에서 실행 중인 코드가 이 페이지에 쓸 수 있음,
UWE 사용자 모드에서 실행 중인 코드가 이 페이지에 쓸 수 있음,
페이지 프레임 번호 V 비트가 세트된 PTE의 경우 이 항목은 그 PTE의 물리적 페이지 프레임 번호를 갖는다. 유효하지 않은 PTE의 경우, 항목의 값이 0이 아니라면 페이지가 스왑 파일 어디에 저장되어 있는지에 대한 정보를 갖고 있다.

리눅스는 다음 두 비트를 정의하여 사용한다:

_PAGE_DIRTY 이 비트가 설정되어 있으면 페이지는 스왑 파일에 기록될 필요가 있다.

_PAGE_ACCESSED 접근된 페이지를 표시하기 위해 리눅스가 사용한다.

3.2 캐시(Cache)

만약 위에서 언급한 이론적 모델을 사용하여 시스템을 구현한다면, 동작하기는 하겠지만 그다지 효율적이지는 않을 것이다. 운영체제와 프로세서 설계자들은 시스템에서 더 많은 성능을 얻어내기 위해 애쓰고 있다. 프로세서, 메모리 등을 더 빠르게 만드는 것 외에, 가장 좋은 방법은 어떤 작업들을 더 빠르게 실행할 수 있도록, 유용한 자료와 데이터의 캐시를 관리하는 것이다. 리눅스는 메모리 관리와 관련하여 몇가지 캐시를 사용한다:

버퍼 캐시(Buffer Cache) 버퍼 캐시는 블럭 디바이스 드라이버가 사용하는 데이터 버퍼들을 갖고 있다. 이들 버퍼는 고정된 크기로(예를 들어 512바이트), 블럭 장치에서 읽거나, 거기에 쓰는 자료의 블럭을 갖고 있다. 블럭 장치는 고정된 크기의 데이터 블럭 단위로 읽기/쓰기만을 할 수 있는 장치이다. 모든 하드 디스크는 블럭 장치이다.

fs/buffer.c 참조

버퍼 캐시는 장치 식별자와 원하는 블럭 번호에 의해 색인되어 있고, 이 색인을 통해 데이터 블럭을 빨리 찾을 수 있다. 블럭 장치는 버퍼 캐시를 통해서만 접근된다. 데이터가 버퍼 캐시에서 발견되면 하드 디스크같은 물리적 블럭 장치에서 읽을 필요가 없으며, 따라서 훨씬 빠르게 접근된다.

페이지 캐시(Page Cache) 페이지 캐시는 디스크상의 이미지와 데이터에 접근하는 속도를 높이기 위해 사용된다. 이것은 파일의 논리적인 내용을 페이지 단위로 캐시하기 위해 사용되며, 파일과 파일 내의 오프셋을 통해 접근된다. 디스크에서 메모리로 페이지들을 읽어들이면, 페이지들은 페이지 캐시에 캐시된다.

mm/filemap.c 참조

스왑 캐시(Swap Cache) 더티 페이지들만이 스왑 파일에 저장된다. 이들 페이지가 스왑 파일에 기록된 다음 더이상 변경되지 않았다면, 그 페이지가 다음에 스왑 아웃될 때는 이미 그 페이지가 (동일한 내용으로) 스왑 파일에 있으므로, 스왑 파일에 기록할 필요가 없다. 대신 그 페이지는 그냥 폐기하면 된다. 스와핑이 심하게 일어나는 시스템에서는 이렇게 함으로써 불필요하고 값비싼 디스크 연산을 많이 줄일 수 있다.

mm/swap_state.c
mm/swapfile. 참조

하드웨어 캐시(Hardware Cache) 흔히 구현되는 하드웨어 캐시는 프로세서 내부에 있는 페이지 테이블 엔트리(PTE)의 캐시이다. 이 경우 프로세서는 항상 페이지 테이블을 직접 읽는 것이 아니라, PTE를 필요로 할 때마다 페이지에 대한 변환 결과를 캐시한다. 이들은 변환 참조 버퍼(Translation Look-aside Buffers, TLB)라고 불리며 시스템의 여러 프로세스의 페이지 테이블 엔트리의 캐시된 복사본을 갖고 있다.

가상 주소를 참조할 때, 프로세서는 TLB 엔트리에서 일치하는 항목을 찾으려고 한다. 만약 찾는다면, 가상 주소를 바로 물리적 주소로 변환하여, 데이터에 대한 올바른 연산을 수행할 수 있다. 프로세서가 일치하는 TLB 엔트리를 찾지 못하면, 운영체제의 도움을 받아야 한다. 도움을 받기 위해 운영체제에게 TLB를 찾지 못했다는(TLB miss) 신호를 보낸다. 문제를 해결하도록 운영체제에게 예외 신호를 전달하기 위해서는 시스템마다 특유한 메커니즘이 사용된다. 운영체제는 주소 변환을 위해 새로운 TLB 엔트리를 생성한다. 예외가 처리된 다음, 프로세서는 같은 가상 주소 변환을 다시 시도한다. 이번에는 이 가상 주소에 해당하는 유효한 TLB 엔트리가 있기 때문에 잘 처리될 것이다.

하드웨어 캐시이든 다른 캐시이든 캐시를 사용하는 것의 단점은, 그렇게 효율을 높이기 위해서 리눅스는 이들 캐시를 관리하는데 더 많은 시간과 공간을 사용해야 한다는 것과, 캐시가 망가지는 경우 시스템이 죽는다는 것이다.

3.3 리눅스 페이지 테이블(Linux Page Table)

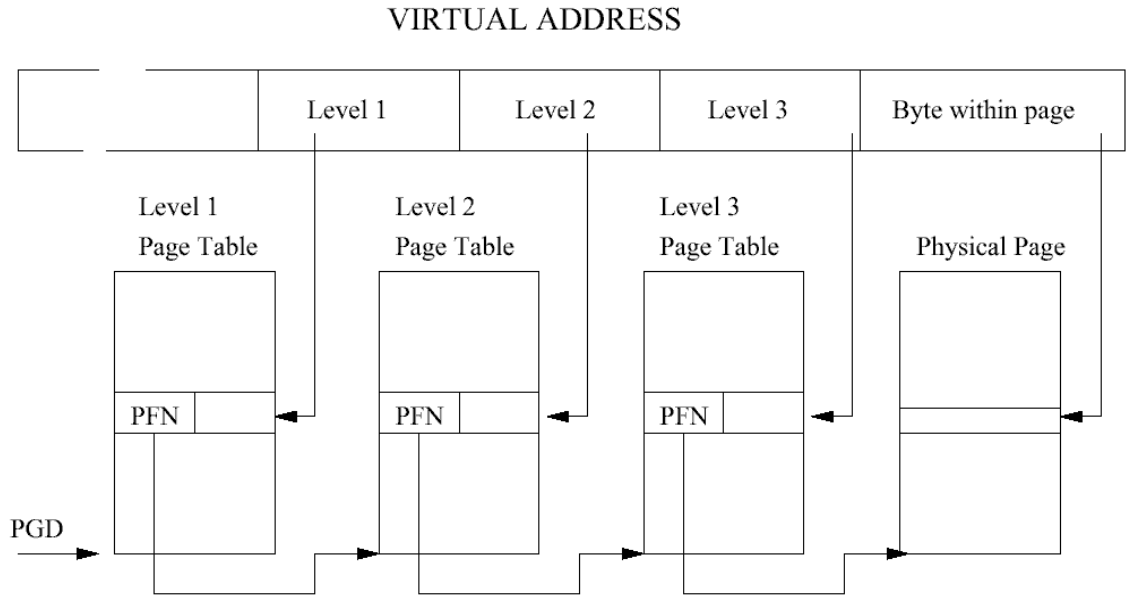


그림 3.3 : 3단계 페이지 테이블

리눅스는 3단계의 페이지 테이블을 가정한다[13]. 접근되는 각 페이지 테이블은 다음 단계의 페이지 테이블의 페이지 프레임 번호를 갖고 있다. 그림 3.3은 가상 주소가 어떻게 여러개의 항목으로 나누어지는지 보여준다. 각 항목은 특정 페이지 테이블에서의 오프셋을 제공한다. 가상 주소를 물리적 주소로 변환하기 위해, 프로세서는 각 단계의 항목의 내용을 가져와서 페이지 테이블을 갖고 있는 물리적 페이지에 대한 오프셋으로 변환하고, 다음 단계의 페이지 테이블의 페이지 프레임 번호를 읽는다. 이 과정을 3회 반복하면 가상 주소를 포함하는 물리적 페이지의 페이지 프레임 번호를 얻을 수 있다. 그리고 가상 주소의 마지막 항목인 바이트 오프셋을 사용하여 페이지 내에 있는 데이터를 얻는다.

include/asm/ pgtable.h 참조

리눅스를 실행하는 플랫폼들은, 반드시 커널이 특정 프로세스의 페이지 테이블을 탐색할 수 있도록 하는 매크로들을 지원해야 한다. 이같은 방식 덕택에 커널은 페이지 테이블 엔트리의 형식이라든가 어떻게 배열되어 있는지 알아야 될 필요가 없다. 이런 방식은 매우 성공적이어서 세단계의 페이지 테이블을 가지는 알파 프로세서와 두 단계의 페이지 테이블을 가지는 인텔의 x86계열의 프로세서에 대해서 동일한 페이지 테이블 처리 코드를 사용하고 있다.

3.4 페이지의 할당(allocation)과 해제(deallocation)

시스템에 있는 물리적 페이지에 대해 여러 요구들이 있다. 예를 들어, 이미지를 메모리에 로드할 때 운영체제는 페이지를 할당해야 있다. 그리고 이미지의 실행이 끝나고 언로드될 때 페이지를 해제해야 한다. 물리적 페이지의 또 다른 용도는 페이지 테이블 자체와 같은 커널 특유의 자료구조를 저장하기 위한 것이다. 페이지 할당과 해제에 사용되는 메커니즘이나 자료구조는, 가상 메모리 서브시스템의 효율성에 가장 중요한 영향을 미친다.

include/linux/ mm.h 참조

시스템의 모든 물리적 페이지는 `mem_map_t`[14] 구조체의 리스트인 `mem_map` 자료구조로 나타내며 이들은 부팅 시에 초기화된다. 각 `mem_map_t` 구조체는 시스템의 물리적 페이지 하나를 기술한다. 메모리 관리에 관해 중요한 항목들은 다음과 같다 :

카운트(count) 이 페이지를 사용하고 있는 사용자(프로세스)들의 수. 페이지를 여러 프로세스

역주 13) 이들은 각기 페이지 디렉토리(page directory), 페이지 중간 디렉토리(page middle directory), 페이지 테이블(page table)이라고 하며, 각기 pgd_t, pmd_t, pte_t 타입으로 정의되어 있다. (flyduck)

14) 헷갈리게도 이 구조체를 페이지(page) 구조체라고도 부른다.

가 공유하고 있다면 카운트는 1보다 크다.

나이(age) 이 항목은 페이지의 나이를 기록하고 있으며, 그 페이지가 폐기 또는 스왑할 좋은 후보인지 결정하는데 사용된다.

map_nr 이 mem_map_t가 기술하는 물리적 페이지의 프레임 번호이다.

free_area 벡터는 페이지를 할당하는 코드가 프리 페이지를 찾는데 사용된다. 전체적인 버퍼 관리 계획은 이런 메커니즘으로 이루어지며 세부적인 코드에 대해서라면, 프로세서가 사용하는 페이지의 크기와 물리적인 페이징 메커니즘은 서로 다를 수 있다.

free_area의 각 원소들은 페이지 블럭들에 대한 정보를 가지고 있다. 배열의 첫번째 원소는 한 페이지를, 그 다음은 두 페이지의 블럭들을, 그 다음은 네 페이지의 블럭들을, 이런식으로 계속 2의 제곱으로 증가하는 페이지의 블럭들을 기술한다. list 원소는 큐의 헤드로 사용되며, mem_map 배열 내의 page 자료구조에 대한 포인터를 갖고 있다. 페이지의 프리 블럭들은 이 큐에 저장된다. map은 이 크기의 할당된 페이지 그룹을 추적하여 관리하는 비트맵에 대한 포인터이다. 비트맵의 비트 N은 페이지의 N번째 페이지 블럭이 프리이면 1로 설정된다.

그림 3.4는 free_area 구조체를 보여준다. 0번째 원소는 하나의 프리 페이지(페이지 프레임 번호 0), 2번째 원소는 두개의 4 페이지 크기의 프리 블럭을 보여준다. 앞의 것은 페이지 프레임 번호 4에서, 뒤의 것은 페이지 프레임 번호 56에서 시작한다.

3.4.1 페이지 할당(Page Allocation)

mm/page_alloc.c __get_free_pages() 참조

리눅스는 페이지 블럭을 효율적으로 할당하고 해제하기 위해 버디 알고리즘(Buddy algorithm)[15]을 사용한다. 페이지 할당 코드는 하나 이상의 물리적 페이지로 구성된 하나의 블럭을 할당한다. 페이지들은 2의 제곱 크기인 블럭으로 할당된다. 즉 1 페이지, 2 페이지, 4 페이지 식으로 블럭을 할당할 수 있다는 것이다. 시스템에 있는 프리 페이지가 요청을 처리하기에 충분하다면(nr_free_pages > min_free_pages), 할당 코드는 free_area에서 요청한 크기에 해당하는 페이지의 블럭을 탐색한다. free_area의 각 원소는 할당된 맵과, 해당 크기를 갖는 페이지의 프리 블럭의 맵을 가지고 있다. 예를 들어 배열의 두번째 원소는, 각각 4 페이지 길이의 할당된 블럭과 프리 블럭을 기술하는 메모리 맵을 가지고 있다.

할당 알고리즘은 먼저 요청된 크기의 페이지 블럭을 검색한다. free_area 자료구조의 list 원소에 큐되어 있는 프리 페이지의 고리를 따라간다. 만일 요청된 크기의 프리 페이지 블럭이 없다면, 그 다음 크기(요청된 크기의 두 배)의 블럭을 찾아본다. 이 과정은 모든 free_area를 다 검색하거나, 사용할 수 있는 페이지 블럭을 찾아낼 때까지 계속된다. 찾아낸 페이지 블럭이 요청한 크기보다 크다면, 그 페이지 블럭은 요청한 크기가 될 때까지 분할한다. 블럭에 들어있는 페이지의 수는 두 배씩 늘어나는 크기로 되어 있기 때문에, 분할과정은 블럭을 반으로 잘라가기만 하면 된다. 프리 블럭은 해당하는 큐에 큐되며 할당된 페이지 블럭은 호출자에게 되돌려진다.

예를 들어, 그림 3.4에서 2 페이지짜리 블럭을 요청했다면, 4 페이지짜리 첫번째 블럭(페이지 프레임 번호 4에서 시작하는)은 2 페이지짜리 블럭 두개로 나뉠 것이다. 프레임 번호 4에서 시작하는 첫번째 블럭은 할당된 페이지가 되어 호출자에게 되돌려지고, 페이지 프레임 번호 6에서 시작하는 두번째 블럭은 2 페이지 크기의 프리 블럭으로 free_area 배열의 첫번째 원소에 있는 큐에 저장된다.

3.4.2 페이지 해제(Page Deallocation)

15) 여기에 참고 목록을 적을 것.

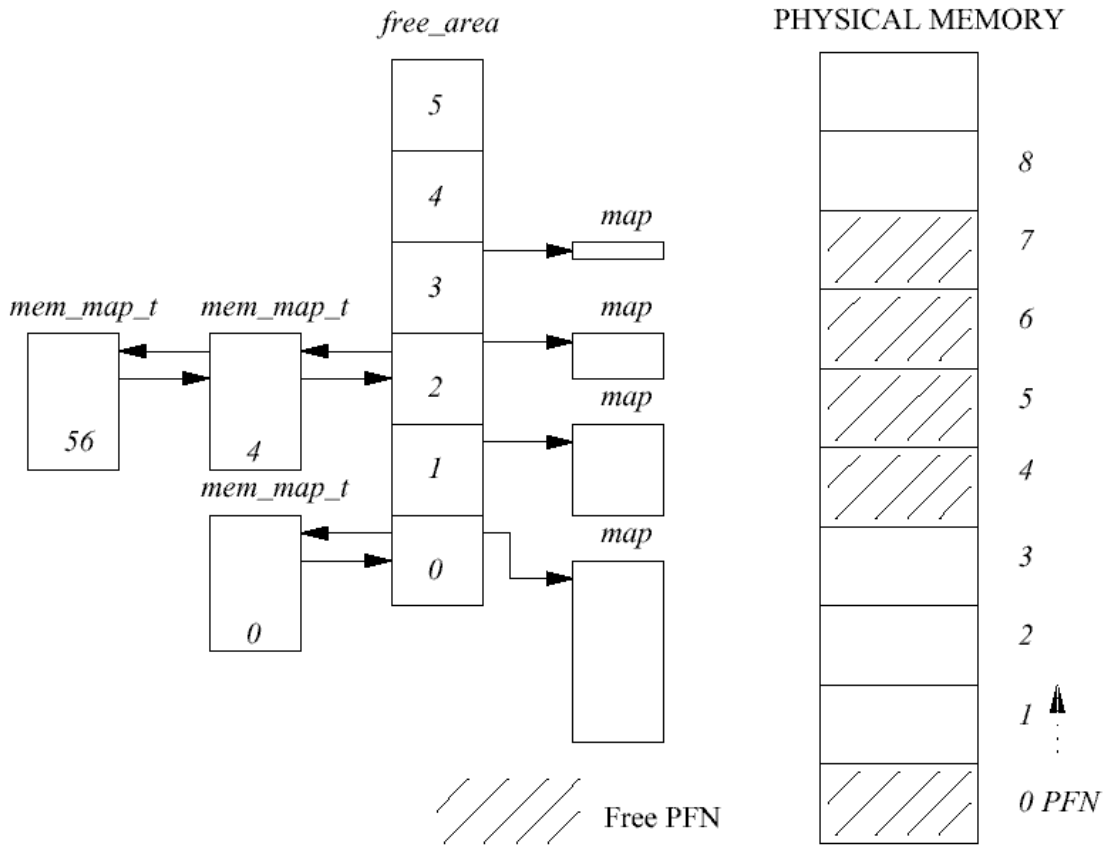


그림 3.4 : free_area 자료구조

mm/page_alloc.c
free_page()
참조

페이지 블럭을 할당하는 것은 더 큰 프리 페이지 블럭을 작은 것으로 쪼개기 때문에 메모리를 조각내게 된다. 페이지 해제 코드는 가능할 때마다 프리 페이지들을 더 큰 블럭의 프리 페이지로 합친다. 사실 페이지 블럭의 크기는 중요한데, 그것이 블럭들을 더 큰 블럭으로 쉽게 합칠 수 있게 하기 때문이다.

페이지 블럭이 해제될 때마다, 같은 크기의 인접한 버디(buddy) 블럭이 프리인지 검사한다. 그렇다면 그 블럭과 새로 프리 블럭이 된 페이지들이 합쳐져서, 새로운 빈 블럭이 되어 다음 크기의 프리 블럭을 이룬다. 두개의 페이지 블럭이 합쳐져서 더 큰 프리 페이지 블럭이 될 때마다, 페이지 해제 코드는 이 블럭을 다시 인접한 것과 합쳐서 더 큰 것으로 만들려고 한다. 이렇게 해서 프리 페이지 블럭은 메모리가 허락하는 만큼 커질 수 있게 된다.

예를 들어, 그림 3.4에서 페이지 프레임 번호 1이 해제되면, 이미 해제되어 있는 페이지 프레임 번호 0과 합쳐져 2페이지 크기의 프리 블럭이 되어, free_area의 첫번째 원소의 큐에 연결된다.

3.5 메모리 매핑(Memory Mapping)

이미지를 실행하려면, 그 실행 이미지의 내용을 프로세스의 가상 주소공간으로 가져와야 한다. 실행 이미지가 링크해서 사용하는 공유 라이브러리도 마찬가지다. 리눅스는 실행파일을 실제로 물리적 메모리에 가져오는 대신에, 단지 프로세스의 가상 메모리와 연결만 시킨다. 그리고 응용 프로그램이 실행되면서 프로그램의 일부가 참조됨에 따라, 실행 이미지로부터 해당하는 이미지 부분을 메모리로 가져온다. 이렇게 이미지를 프로세스의 가상 주소공간에 연결하는 것을 메모리 매핑이라고 한다.

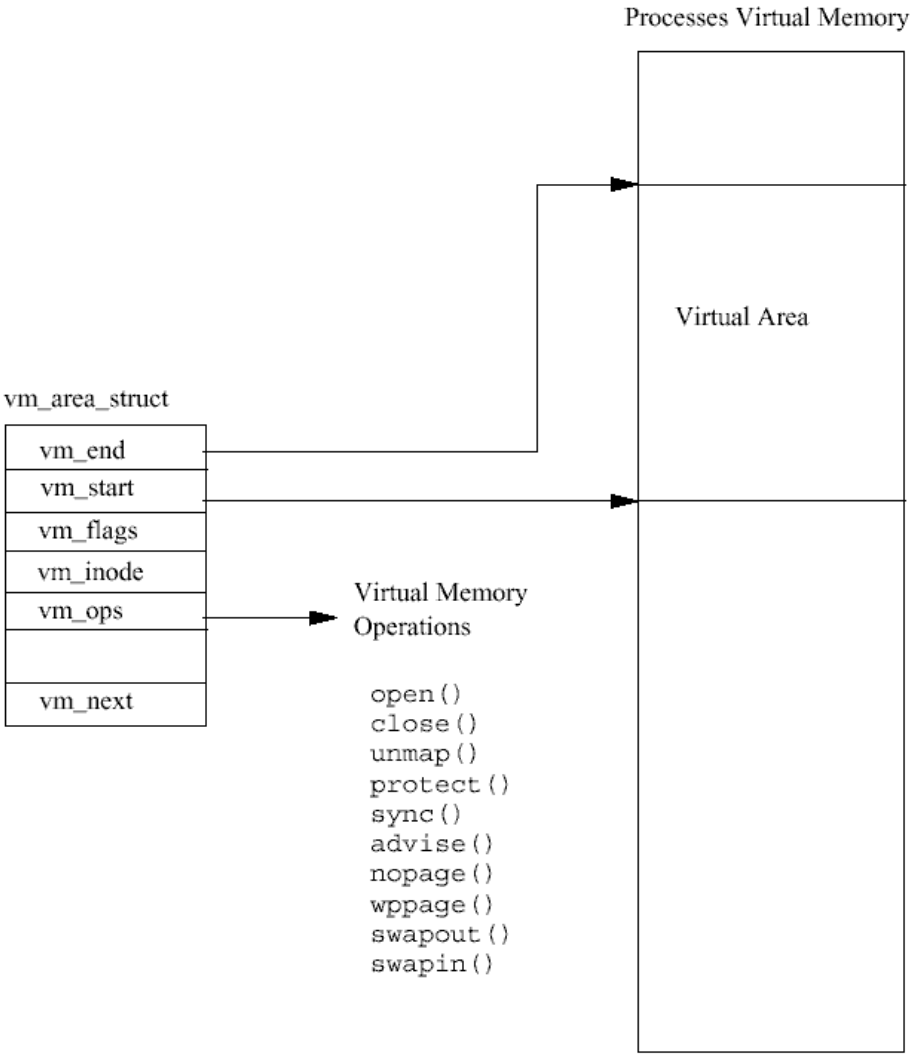


그림 3.5 : 가상 메모리의 영역들

모든 프로세스의 가상 메모리는 `mm_struct` 자료구조로 표현된다. 여기에는 현재 실행중인 이미지(예를 들어, `bash`의)에 대한 정보와, 여러개의 `vm_area_struct` 자료구조에 대한 포인터가 들어 있다. 각각의 `vm_area_struct` 자료구조는 가상 메모리 영역의 시작과 끝, 프로세스의 접근 권한, 메모리에 대한 연산들 등을 기술한다. 여기서 연산은 이 영역의 가상 메모리를 처리하기 위해 리눅스가 사용해야 하는 루틴들이다. 예를 들어, 가상 메모리 연산 중의 하나는, 프로세스가 가상 메모리를 접근하려다 (페이지 폴트를 통해) 그 메모리가 실제로는 물리적 메모리에 없다는 것을 알았을 때, 이를 처리하는 올바른 작업을 수행한다. 이 연산이 *nopage* 연산이다. 리눅스는 실행 이미지의 페이지를 메모리로 옮길 것을 요구할 때 *nopage* 연산을 사용한다.

어떤 실행 이미지가 프로세스의 가상 주소에 매핑될 때, 한 세트의 `vm_area_struct` 자료구조가 만들어진다. 각 `vm_area_struct` 자료구조는 실행 이미지의 한 부분을 나타낸다 - 실행 코드, 초기화된 데이터(변수), 초기화되지 않은 데이터(BSS) 등이다. 리눅스는 상당수의 표준 가상 메모리 연산을 지원하며, `vm_area_struct` 자료구조가 만들어질 때, 그에 맞는 일련의 가상 메모리 연산이 여기에 지정된다.

3.6 요구 페이징(Demand Paging)

실행 이미지가 프로세스의 가상 메모리에 매핑되고 나면, 실행할 수 있게 된다. 이미지의 맨

mm/memory.c handle_mm_fault () 참조

앞부분만 물리적으로 메모리에 올라와 있기 때문에, 곧 아직 물리적 메모리에 있지 않은 가상 메모리 영역을 접근하게 된다. 프로세스가 유효한 페이지 테이블 엔트리를 갖지 않은 가상 주소에 접근하면, 프로세서는 리눅스에 페이지 폴트를 보고한다. 페이지 폴트는 페이지 폴트가 발생한 페이지와, 페이지 폴트를 발생시킨 메모리 접근의 유형을 설명한다.

리눅스는 페이지 폴트가 발생한 곳을 포함하는 메모리 영역을 나타내는 `vm_area_struct`를 찾아야 한다. `vm_area_struct` 자료구조를 검색하는 것은, 페이지 폴트를 효율적으로 처리하는데 있어 핵심적이기 때문에, 이들 자료구조는 AVL(Adelson-Velskii and Landis)[16]트리 구조로 만들어져 있다. 만약 폴트가 발생한 가상 주소에 대한 `vm_area_struct` 자료구조가 없다면, 이 프로세스는 금지된 가상 주소에 접근한 것이다. 리눅스는 SIGSEGV[17] 시그널을 이 프로세스에 보내며, 이 프로세스가 그 시그널을 처리하는 핸들러를 갖고 있지 않다면, 프로세스는 종료될 것이다.

그런다음 리눅스는 발생한 페이지 폴트의 유형과, 가상 메모리의 이 영역에 대해 허용된 접근 유형을 비교한다. 프로세스가 읽기만 허용된 영역에 쓰려고 하는 것처럼, 허용되지 않은 방법으로 접근하려고 하면 메모리 에러가 시그널로 전달된다.

mm/memory.c do_no_page() 참조

페이지 폴트가 올바른 것이라도 판단했다면, 리눅스는 이를 처리해야 한다. 리눅스는 스왑 파일에 있는 페이지와, 디스크의 어딘가에 있는 실행 이미지의 일부인 페이지를 구분해야 한다. 구분을 위해 폴트가 발생한 가상 주소의 페이지 테이블 엔트리를 사용한다.

그 페이지의 페이지 테이블 엔트리가 유효하지 않지만 비어있지도 않다면, 페이지 폴트는 스왑 파일에 들어있는 페이지에 대하여 발생한 것이다. 알파 AXP의 페이지 테이블이라면, 유효 비트가 설정되지 않고, PFN 항목에 0이 아닌 값을 가진 엔트리들이 이에 해당된다. 이 경우 PFN 항목은 스왑 파일의(그리고 어떤 스왑 파일의) 어느 부분에 그 페이지가 들어있는지에 대한 정보를 갖고 있다. 스왑 파일에 있는 페이지들을 어떻게 다루는가는 이 장의 뒤에서 설명한다.

mm/filemap.c filemap_nopage () 참조

모든 `vm_area_struct` 자료구조가 가상 메모리 연산을 갖고 있는 것은 아니고, 가지고 있다고 해도 *nopage*연산을 가지고 있지 않을 수도 있다. 이는 기본적으로 리눅스가 새로운 물리적 페이지를 할당하고 이에 대한 유효한 페이지 테이블 엔트리를 생성하여, 이를 처리해 주기 때문이다. 이 가상 메모리 영역 용으로 *nopage* 연산이 있다면, 리눅스는 이를 사용할 것이다.

일반 *nopage* 연산은 메모리에 매핑된 실행 이미지를 위해 사용되며, 페이지 캐시를 사용하여 요청한 페이지를 실제 메모리로 가져온다.

어쨌든 요청한 페이지가 물리적 메모리로 올라오면, 프로세스의 페이지 테이블이 갱신된다. 이 엔트리를 갱신하기 위하여, 특히 변환 참조 버퍼(translation look aside buffer)를 사용하는 프로세서의 경우에는, 특정한 하드웨어에 맞는 행동이 필요할 수도 있다. 이제 페이지 폴트가 처리되었으므로 그 상황은 해제되며, 프로세스는 가상 메모리 접근에 대한 폴트를 발생시켰던 명령에서부터 실행을 재개한다.

3.7 리눅스 페이지 캐시

include/linux/ pagemap.h 참조

리눅스 페이지 캐시의 역할은 디스크에 있는 파일로의 접근 속도를 높이는 것이다. 메모리 매핑된 파일은 한번에 한 페이지씩 읽혀지며, 이들 페이지는 페이지 캐시에 저장된다. 그림 3.6은 페이지 캐시가 `mem_map_t` 자료구조에 대한 포인터들의 벡터인 `page_hash_table`로 구성되어 있는 것을 보여준다. 리눅스의 각 파일은 VFS `inode` 자료구조(9장, 파일 시스템에서 설명)에 의해 식별되며, 각 VFS `inode`는 유일하며, 한 파일에 일대일로 대응되어 파

역주 16) 높이 균형을 이루는 이진 트리, 사실 이 사람 이름들은 몰라도 된다. (심마로)
역주 17) Segmentation Fault. (flyduck)

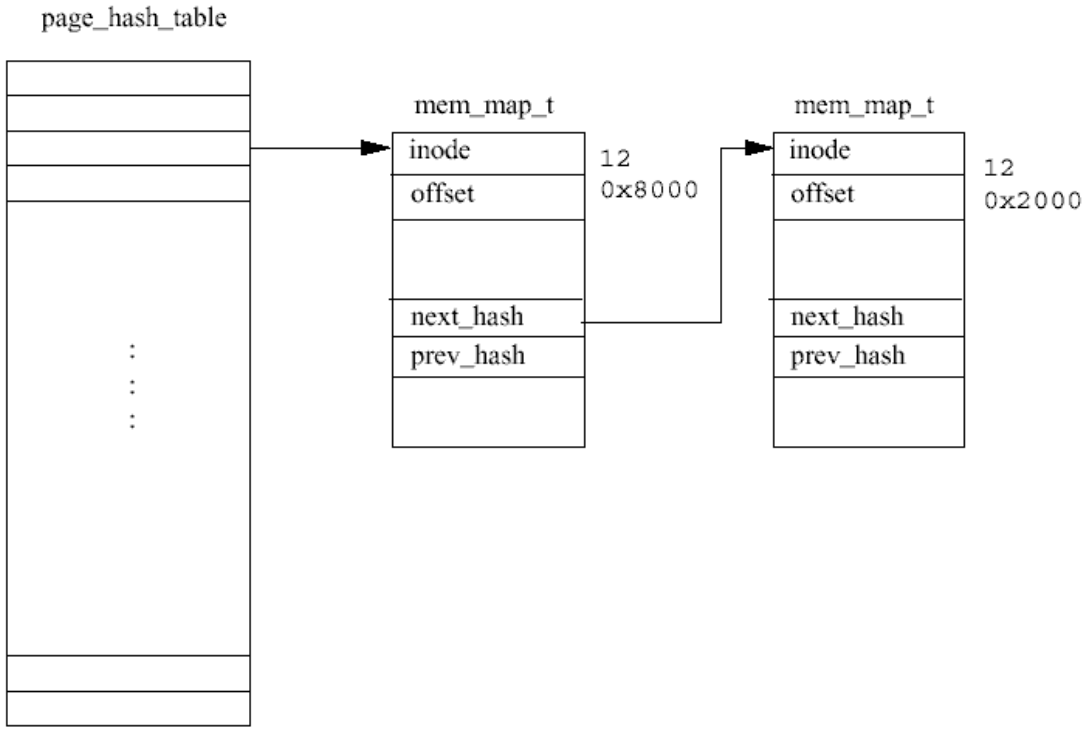


그림 3.6 : 리눅스 페이지 캐시

일을 완전히 기술한다. 페이지 테이블에 대한 인덱스는, 파일의 VFS `inode`와 파일에서의 오프셋을 가지고 만들어진다.

페이지를 메모리 매핑된 파일에서 읽을 때, 예를 들어 요구 페이징에서 페이지를 메모리로 다시 가져올 때, 페이지는 페이지 캐시를 통해 읽게 된다. 페이지가 캐시에 있으면, 그 페이지를 나타내는 `mem_map_t` 자료구조에 대한 포인터가 페이지 폴트 처리 코드로 되돌려진다. 캐시에 없다면 이미지를 갖고 있는 파일 시스템으로부터 페이지를 메모리로 가져와야 한다. 리눅스는 물리적 페이지를 할당하고 디스크 상의 파일로부터 페이지를 읽어 들인다.

가능하다면 리눅스는 파일의 다음 페이지에 대한 읽기를 시작한다. 이렇게 한 페이지를 미리 읽는 것은, 프로세스가 파일의 페이지를 순차적으로 접근하는 경우, 다음 페이지가 (프로세스가 다음 메모리를 읽기 전에) 프로세스를 위한 메모리에 기다리고 있게 한다.

시간이 흘러 이미지를 읽고 실행함에 따라 페이지 캐시가 증가하게 된다. 페이지는 더이상 필요없게 되면, 가령 이미지가 더이상 어떤 프로세스에 의해서도 사용되지 않게 되면, 캐시로부터 제거된다. 리눅스가 메모리를 사용해 나감에 따라 물리적 페이지가 부족해지기 시작한다. 이 때 리눅스는 페이지 캐시의 크기를 줄일 것이다.

3.8 페이지의 스왑 아웃(swap out)과 폐기(discarding)

물리적 메모리가 부족하게 되면 리눅스 메모리 관리 서브시스템은 물리적 메모리를 해제하려 한다. 이 일은 커널 스왑 데몬(`kswapd`)에게 할당된다. 커널 스왑 데몬은 커널 쓰레드라는 특별한 종류의 프로세스이다. 커널 쓰레드는 가상 메모리 없이, 물리적 메모리 공간에서 커널모드로 실행되는 프로세스이다. 커널 스왑 데몬이라는 이름은 약간 잘못되었는데, 이는 단지 페이지를 스왑 아웃하여 시스템의 스왑 파일에 저장하는 것 이상의 여러 일을 하기 때문이다. 커널 스왑 데몬의 역할은 메모리 관리 시스템이 효율적으로 동작할 수 있도록 시스템에 충분한 프리 페이지가 있도록 하는 것이다.

mm/vmscan.c
kswapd() 참조

커널 스왑 데몬(kswapd)은 커널의 init 프로세스에 의해 시작되며 커널 스왑 타이머가 주기적으로 만료될 때를 기다리고 있다. 타이머가 만료될 때마다, 스왑 데몬은 시스템의 프리 페이지 수가 너무 적지 않은지 확인한다. free_pages_high와 free_pages_low라는 두개의 변수를 사용하여, 페이지를 해제해야 할 필요가 있는지 결정한다. 시스템에 남아있는 프리 페이지의 수가 free_pages_high보다 큰 동안은, 커널 스왑 데몬은 아무 일도 하지 않고 다시 잠들어 다음 타이머가 만료될 때를 기다린다. 이 확인 작업을 위해, 커널 스왑 데몬은 현재 스왑 파일에 씌어지고 있는 페이지의 수도 고려한다. 이 개수는 nr_async_pages라는 카운트 값으로 유지된다. 이 값은 어떤 페이지가 스왑 파일에 씌어지기 위해 큐에 들어갈 때마다 증가하고, 스왑 장치에 완전히 씌어질 때마다 감소한다. free_pages_low와 free_pages_high는 시스템이 부팅할 때 설정되며, 시스템에 있는 실제 페이지 수와 관련이 있다. 만약, 시스템에 있는 프리 페이지 수가 free_pages_high보다, 심지어는 free_pages_low보다 작아지면, 커널 스왑 데몬은 시스템이 사용하는 물리적 페이지의 수를 줄이기 위하여 다음 세가지 방법을 시도한다.

버퍼 캐시와 페이지 캐시의 크기를 줄인다.

시스템 V 공유 메모리 페이지를 스왑 아웃한다.

페이지를 스왑 아웃하고 폐기한다.

시스템의 프리 페이지의 수가 free_pages_low 이하로 떨어지면, 커널 스왑 데몬은 다음에 실행되기 전에, 6개의 페이지를 해제하려 한다. 그렇지 않으면 3개의 페이지를 해제하려고 한다. 충분한 페이지들이 해제될 때까지 위의 각 방법이 차례로 시도된다. 커널 스왑 데몬은 물리적 페이지를 해제하기 위해 지난번에 어떤 방법을 사용했는지 기억하고, 매번 실행될 때마다 최종적으로 성공한 방법을 사용해서 페이지를 해제시키려고 한다[18].

충분한 페이지를 해제한 후, 스왑 데몬은 다시 잠들어 타이머가 만료되길 기다린다. 커널 스왑 데몬이 페이지를 해제한 이유가, 프리 페이지의 수가 free_pages_low 이하로 떨어져서 였다면, 평소에 자던 시간의 절반만 잔다. 그래서 빈 페이지의 수가 free_pages_low보다 커지면 커널 스왑 데몬은 더 오랬동안 자게 된다.

3.8.1 페이지 캐시와 버퍼 캐시 크기를 줄이기

페이지 캐시와 버퍼 캐시에 들어있는 페이지는 free_area 벡터로 해제할 좋은 후보들이다. 메모리에 매핑된 파일의 페이지를 갖고 있는 페이지 캐시는 시스템의 메모리를 채우고 있는 불필요한 페이지를 갖고 있을 수 있다. 마찬가지로 실제 장치로 쓰거나 읽은 데이터 버퍼를 갖고 있는 버퍼 캐시 역시 불필요한 버퍼를 갖고 있을 수 있다. 시스템의 실제 페이지가 고갈되기 시작하면, 이들 캐시로부터 페이지를 버리는 것은, 메모리에서 스왑 아웃하는 경우와 달리 실제 장치에 기록할 필요가 없으므로 상대적으로 쉽다. 이들 페이지를 버리는 것은 실제 장치와 메모리 매핑된 파일을 액세스하는 속도가 느려진다는 것을 제외하고는 다른 심각한 부작용은 없다. 그리고 이들 캐시로부터 페이지를 제거하는 것이 공정하게 이루어진다면, 모든 프로세스들은 공평하게 손해볼 것이다.

mm/filemap.c
shrink_mmap()
참조

커널 스왑 데몬이 이들 캐시를 줄이려고 할 때 마다, mem_map 페이지 벡터에 있는 페이지 블럭을 검사하여 실제 메모리에서 버려도 될 것이 있는지 확인한다. 커널 스왑 데몬이 심하게 스와핑을 하고 있다면 - 즉, 시스템의 프리 페이지의 수가 심각하게 낮게 떨어졌다면 - 검사할 페이지 블럭의 크기가 커진다. 페이지 블럭은 돌아가며 검사된다. 즉 메모리 맵을 줄이려고 할 때마다 서로 다른 페이지 블럭이 검사된다. 이 방법은 시계 알고리즘(clock algorithm)이라고 불리는데, 시계 바늘의 움직임처럼 전체 mem_map 페이지 벡터에서 한번에

역주 18) 이 밖에 min_free_pages 라는 값이 있는데, 이는 커널이 필요로 하는 경우 바로 프리 페이지를 얻을 수 있도록, 프리 페이지의 갯수가 이 값 이하로 떨어지지 않도록 한다. 이 값 역시 부팅시에 설정이 된다. (flyduck)

몇 페이지씩 차례로 조사되기 때문이다.

조사되는 각 페이지는 그것이 페이지 캐시나 버퍼 캐시에 있는 것인지 검사된다. 이 단계에서 공유 페이지는 고려되지 않으며, 한 페이지가 동시에 두 캐시에 모두에 있을 수 없다는 것을 기억해 두기 바란다. 페이지가 두 캐시 어디에도 속하지 않으면 `mem_map` 페이지 벡터의 다음 페이지가 조사된다.

버퍼의 할당과 해제가 더욱 효율적으로 이루어지게 하기 위하여 (페이지 내의 버퍼가 캐시되는 것이 아니라) 페이지 자체가 버퍼 캐시에 캐시된다. 메모리 맵 축소 코드는 검사되는 페이지에 포함된 버퍼를 해제하려고 한다. 페이지에 포함된 모든 버퍼가 해제되면, 그들을 갖고 있던 페이지도 해제된다. 조사된 페이지가 리눅스 페이지 캐시에 있다면, 페이지 캐시에서 제거된 다음 해제된다.

fs/buffer.c try_to_free_buffer() 참조

이렇게 해서 충분한 페이지가 해제되었다면 커널 스왑 데몬은 다음에 주기적으로 깨어나는 시점까지 기다린다. 해제되는 페이지 중에는 어떤 프로세스의 가상 메모리에도 속하지 않으므로 (모두 캐시된 페이지이므로), 아무런 페이지 테이블도 수정할 필요가 없다. 캐시된 페이지를 제거하는 걸로 충분하지 않은 경우, 스왑 데몬은 공유 페이지를 스왑 아웃하려고 하게 된다.

3.8.2 시스템 V 공유 메모리 페이지의 스왑 아웃

시스템 V 공유 메모리는 둘 이상의 프로세스가 가상 메모리를 공유하여 그들 사이에 정보를 전송할 수 있는 프로세스간 통신(IPC) 메커니즘의 일종이다. 프로세스들이 이 방법으로 어떻게 메모리를 공유하는가는 5장에서 자세히 설명한다. 아직은 시스템 V 공유 메모리의 각 영역을 `shmid_ds` 자료구조로 기술한다고 알아두는 것으로 충분하다. 이 자료구조는 이 가상 메모리 영역을 공유하는 프로세스마다 하나씩 대응되는 `vm_area_struct` 자료구조 리스트에 대한 포인터를 갖고 있다. `vm_area_struct` 자료구조는 각 프로세스의 가상 메모리의 어디에 이 시스템 V 공유 메모리가 대응하는지 나타낸다. 이 시스템 V 공유 메모리용 `vm_area_struct` 자료구조들은 `vm_next_shared`, `vm_prev_shared` 포인터로 서로 연결되어 있다. 각각의 `shmid_ds` 자료구조는 이밖에 공유 가상 페이지가 매핑되어 있는 실제 페이지를 설명하고 있는 페이지 테이블 엔트리의 리스트도 갖고 있다.

커널 스왑 데몬은 시스템 V 공유 메모리 페이지를 스왑 아웃할 때에도 시계 알고리즘(clock algorithm)을 사용한다. 커널 스왑 데몬은 실행할 때마다 맨 마지막으로 스왑 아웃한 공유 가상 메모리 페이지가 무엇이었는지를 기억한다. 이를 위해 두개의 인덱스 값을 유지 하는데, 하나는 `shmid_ds` 자료구조 집합에 대한 인덱스이고, 다른 하나는 시스템 V 공유 메모리 영역을 나타내는 페이지 테이블 엔트리의 리스트에 대한 인덱스이다. 이 방법은 시스템 V 공유 메모리 영역이 공정하게 희생되게 한다.

ipc/shm.c shm_swap() 참조

어떤 시스템 V 공유 메모리의 가상 페이지에 대한 물리적 페이지 프레임 번호는, 이 가상 메모리 영역을 공유하는 모든 프로세스의 페이지 테이블에 들어있기 때문에, 커널 스왑 데몬은 이들 페이지 테이블 모두를 변경하여, 이 페이지가 더이상 메모리에 없고 스왑 파일에 들어 있다는 것을 알려주어야 한다. 스왑 아웃되는 각 공유 페이지마다, 커널 스왑 데몬은 이 페이지를 공유하고 있는 프로세스들의 페이지 테이블로부터 페이지 테이블 엔트리를 찾는다 (각 `vm_area_struct` 자료구조에서 포인터를 따라감으로써). 이 시스템 V 공유 메모리 페이지에 대한 프로세스의 페이지 테이블 엔트리가 유효하면, 데몬은 그것을 유효하지 않고 스왑 아웃된 페이지 테이블 엔트리로 변환하고, 이 (공유된) 페이지의 사용자 수를 1 감소시킨다. 스왑 아웃된 시스템 V 공유 페이지 테이블 엔트리에는, `shmid_ds` 자료구조 집합에 대한 인덱스와, 이 시스템 V 공유 메모리 영역에 대한 페이지 테이블 엔트리의 인덱스가 들어 있다.

공유하는 프로세스들의 페이지 테이블이 모두 변경되어 그 페이지의 카운트가 0이 되면, 이 공유 페이지를 스왑 파일로 스왑 아웃할 수 있게 된다. 이 시스템 V 공유 메모리 영역에 대

한 shmid_ds 자료구조가 가리키고 있는 리스트에 들어 있는 페이지 테이블 엔트리들은 스왑 아웃된 페이지 테이블 엔트리로 교체된다. 스왑 아웃된 페이지 테이블 엔트리는 유효하지 않지만, 열린 스왑 파일들 중 하나를 가리키는 인덱스와, 그 파일 안의 어디에 스왑 아웃된 페이지가 있는지를 나타내는 오프셋을 갖고 있다. 이 정보는 그 페이지를 다시 물리적 메모리로 가져올 때 사용된다.

3.8.3 페이지의 스왑 아웃과 폐기

mm/vmscan.c swap_out() 참조

스왑 데몬은 시스템에 있는 각 프로세스를 차례로 관찰하면서, 그것이 스왑하기 좋은 후보인지 판단한다. 좋은 후보는 스왑될 수 있으면서(스왑될 수 없는 프로세스도 있다), 메모리에서 스왑되거나 폐기될 수 있는 페이지를 하나 이상 가진 프로세스들이다. 페이지들은 그 안에 저장된 데이터를 다른 방법으로 얻어올 수 있는 방법이 없을 때만, 물리적 메모리로부터 시스템의 스왑 파일에 스왑 아웃된다.

이를 위해 프로세스의 mm_struct 에 큐된 vm_area_struct 자료구조 리스트에 있는 vm_next 포인터를 따라간다.

실행 이미지의 상당수는 실행 파일에서 가져온 것이며, 그 파일에서 쉽게 다시 읽을 수 있다. 예를 들어 이미지에 들어있는 실행 명령은 변경되지 않기 때문에 스왑 파일에 쓸 필요가 없다. 이들 페이지는 그냥 폐기하고, 프로세스가 이들을 다시 참조할 때, 실행 이미지에서 메모리에 다시 가져오게 된다.

스왑할 프로세스를 결정하면, 스왑 데몬은 그 프로세스의 가상 메모리 영역을 전부 보면서 공유되거나 락이 걸리지 않은 영역을 찾는다. 리눅스는 선택된 프로세스에 있는 스왑 가능한 페이지를 모두 스왑 아웃하지는 않는다. 대신 페이지 몇 개만 제거할 뿐이다. 메모리에 락되어 있는 페이지는 스왑하거나 폐기할 수 없다.

mm/vmscan.c swap_out_vma() 참조

리눅스 스왑 알고리즘은 페이지 에이징(page aging)을 사용한다. 각 페이지는 카운터를 가지고 있어서 (mem_map_t 자료구조에 저장되어 있다), 커널 스왑 데몬이 어떤 페이지를 스왑하는 것이 좋은지 결정하는데 도움을 준다. 페이지는 사용하지 않으면 나이를 먹고, 사용할수록 젊어진다; 스왑 데몬은 나이가 많은 페이지만을 스왑 아웃한다. 페이지를 처음 할당할 때 페이지의 초기 나이는 3이다. 페이지가 사용될 때마다, 나이값은 3씩 증가되어 최대 20까지 증가된다(이 값이 작을수록 오래된 페이지이다). 커널 스왑 데몬이 실행될 때마다 페이지의 나이값을 1씩 감소시켜 페이지를 오래된 것으로 만든다. 이 기본 동작은 변경될 수 있으며, 이런 이유로 (다른 스왑 관련 정보와 함께) swap_control 자료구조에 저장되어 있다.

페이지가 아주 오래되면 (나이가 0이 되면) 스왑 데몬은 그 페이지를 좀 더 처리하게 된다. 더티 페이지는 스왑 아웃될 수 있는 페이지이다. 리눅스는 PTE에서 아키텍쳐 특유의 비트를 사용해서 페이지를 이와 같은 방식으로 기술한다 (그림 3.2 참조) 그러나, 모든 더티 페이지가 반드시 스왑 파일에 기록되어야 하는 것은 아니다. 어떤 프로세스는 모든 가상 메모리 영역에서 자신의 스왑 연산(vm_area_struct의 vm_ops 포인터가 가리킴)을 가질 수 있으며, 이 경우 그 연산이 사용된다[19]. 연산이 정의되지 않았다면 스왑 데몬은 스왑 파일에 페이지를 할당하고 스왑 페이지를 스왑 파일에 기록한다.

이제 그 페이지의 페이지 테이블 엔트리는 유효하지 않다고 표시되었지만, 여기에는 이 페이지가 스왑 파일의 어디에 저장되었는지에 대한 정보가 들어 있다. 이 정보는 어느 스왑 파일이 사용되었는지, 그리고 스왑 파일 내에서 페이지가 저장된 위치의 오프셋으로 구성된다. 어떤 스왑 방법을 사용하였든, 원래의 물리적 페이지는 다시 free_area에 넣어져서 프리 상태가 된다. 클린 페이지(더티하지 않은 페이지)는 폐기되어 재사용할 수 있도록

역주 19) vm_area_struct 자료구조에는 해당 가상 메모리 영역에 대한 연산을 할 때 사용할 함수들에 대한 포인터가 들어 있다. 이것이 NULL 값이라면 기본 동작을 수행하지만, 따로 지정된 것이 있다면 해당하는 함수를 부르게 된다. swapout이나 swapin 함수가 여기에 들어있으며, 이전에 설명한 nopage 연산도 여기에 함수 포인터로 들어있다. 여기서는 swapout 연산에 대한 포인터가 사용된다. include/linux/mm.h의 struct vm_area_struct, struct vm_operations_struct 참조. (flyduck)

free_area에 들어간다.

스왑 가능한 프로세스 페이지를 충분히 스왑 아웃하거나 폐기하면, 스왑 데몬은 다시 잠든다. 스왑 데몬이 다음에 깨어났을 때는, 시스템의 다음 프로세스를 검토하게 된다. 이런 방식으로 스왑 데몬은 시스템이 다시 균형에 이를 때까지 각 프로세스의 물리적 페이지를 조금씩 없앤다. 이것은 전체 프로세스를 스왑 아웃하는 것보다 훨씬 공정하다.

3.9 스왑 캐시(Swap Cache)

리눅스는 페이지를 스왑 파일에 스왑 아웃할 때, 페이지를 쓸 필요가 없을 땐 쓰지 않으려고 한다. 어떤 페이지가 스왑 파일과 물리적 메모리에 (같은 내용으로) 동시에 존재하는 경우가 있다. 이런 경우는 어떤 페이지가 메모리에서 스왑 아웃되었다가, 한 프로세스가 그 페이지에 다시 접근하여 메모리로 다시 들어온 경우에 발생한다. 이 때 메모리상의 페이지가 덮어 씌어지지 않는 한 스왑 파일에 있는 페이지의 복사본은 유효하다.

리눅스는 이러한 페이지들을 추적하기 위해 스왑 캐시를 사용한다. 스왑 캐시는 페이지 테이블 엔트리의 리스트로, 각 엔트리는 시스템에 있는 물리적 페이지 하나에 해당한다. 이 페이지 테이블 엔트리는 하나의 스왑 아웃 페이지에 대한 것으로, 그 페이지가 어느 스왑 파일에, 어느 위치에 있는지를 기술한다. 만약 스왑 캐시 엔트리 값이 0이 아닌 경우, 변경되지 않은 페이지가 스왑 파일 내에 들어 있다는 것을 나타낸다. 페이지가 (덮어 씌어져서) 변경된 경우, 그 페이지의 엔트리는 스왑 캐시에서 삭제된다.

리눅스가 어떤 물리적 페이지를 스왑 파일에 스왑 아웃할 필요가 있을 때, 먼저 스왑 캐시에 문의하며, 만약 이 페이지에 대한 유효한 엔트리가 있는 경우, 이 페이지는 스왑 파일에 기록할 필요가 없다. 왜냐하면 메모리에 있는 페이지의 내용이 스왑 파일로부터 마지막으로 읽은 다음 한번도 변경되지 않았기 때문이다.

스왑 캐시의 엔트리는 스왑 아웃된 페이지에 대한 페이지 테이블 엔트리이다. 이들은 유효하지 않다고 표시되어 있지만, 리눅스가 올바른 스왑 파일과 그 스왑 파일 내에서의 올바른 페이지를 찾을 수 있도록 하는 정보를 갖고 있다.

3.10 페이지 스왑 인(Swapping Pages In)

응용 프로그램이 이미 스왑 아웃된 물리적 페이지에 있는 가상 메모리에 쓰려고 하는 경우처럼 스왑 파일에 저장된 더티 페이지들이 다시 필요로 한 경우가 있다. 물리적 메모리에 있지 않은 페이지에 접근하면 페이지 폴트가 발생한다. 페이지 폴트는 프로세서가 가상 주소를 물리적 주소로 변환할 수 없을 때 운영체제에 보내는 신호이다. 이 경우는 가상 메모리 페이지가 스왑 아웃되었을 때에는 이 페이지를 기술하는 페이지 테이블 엔트리가 유효하지 않다고 표시되기 때문에 페이지 폴트가 발생하는 것이다. 프로세서는 가상 주소를 물리적 주소로 변환할 수 없기에, 제어를 운영체제에 넘겨주면서 폴트가 발생한 가상 주소와 폴트의 이유를 알린다. 이 정보의 형식과 프로세서가 운영체제에 제어를 넘기는 방법은 프로세서에 따라 다르다. 프로세서마다 다르게 구현되어 있는 페이지 폴트를 처리하는 코드는 폴트가 발생한 가상 주소를 포함하고 있는 가상 주소 영역을 나타내는 vm_area_struct 자료구조를 찾아야 한다. 이 코드는 폴트가 발생한 가상 주소가 들어있는 자료구조를 찾을 때까지, 해당 프로세스가 사용하는 vm_area_struct 자료구조를 검색한다. 이 작업은 매우 짧은 시간 안에 이루어져야 하므로, 프로세스들이 가지고 있는 vm_area_struct 자료구조는 이 검색을 가능한 빨리 할 수 있도록 배치되어 있다[20].

arch/i386/mm/fault.c do_page_fault() 참조

역주 20) 앞에서 설명한 바와 같이 프로세스에 관련된 메모리를 나타내는 mm_struct에는 vm_area_struct의 연결 리스트와 함께 AVL 트리를 같이 가지고 있다. AVL 트리를 관리하는 것은 약간의 오버헤드가 있지만 페이지 폴트를 빨리 처리하기 위해서는 이를 감수해야 한다. (flyduck)

mm/memory.c
do_no_page()
참조

프로세서에 따라 적절한 작업을 수행하여 폴트가 발생한 가상 주소가 가상 메모리의 유효 영역이라고 판단하면, 페이지 폴트 처리는 이제 일반화되어 리눅스가 동작하는 모든 프로세서에 적용되는 코드로 넘어가게 된다. 일반화된 페이지 폴트 처리 코드는 폴트가 발생한 가상 주소에 대한 페이지 테이블 엔트리를 찾는다. 찾은 페이지 테이블 엔트리가 스왑 아웃된 페이지를 가리키고 있으면, 리눅스는 그 페이지를 다시 물리적 메모리로 가져와야 한다. 스왑 아웃된 페이지에 대한 페이지 테이블 엔트리의 형식은 프로세서마다 다르지만, 어쨌든 모든 프로세서들은 이 페이지가 유효하지 않다고 표시하고, 스왑 파일에서 페이지의 위치를 찾는데 필요한 정보를 페이지 테이블 엔트리에 넣어두고 있다. 리눅스는 페이지를 다시 물리적 메모리로 가져오기 위해 이 정보를 필요로 한다.

mm/memory.c
do_swap_page()
참조

ipc/shm.c
shm_swap_in()
참조

mm/page_alloc.c
swap_in() 참조

이 시점에서, 리눅스는 폴트가 발생한 가상 주소와, 이 페이지가 어디에 스왑되어 있는지에 대한 정보를 갖고 있는 페이지 테이블 엔트리를 알고 있다. `vm_area_struct` 자료구조는 자신이 기술하는 가상 메모리 영역의 어떤 페이지를 물리적 메모리로 스왑할 수 있는 루틴에 대한 포인터를 가지고 있을 수 있다. 이것이 `swapin` 연산이다[21]. 이 가상 메모리 영역에 대해 `swapin` 연산이 정의되어 있으면 리눅스는 그것을 사용한다. 사실 시스템 V 공유 메모리의 스왑 아웃이 이렇게 처리되는데, 스왑 아웃된 시스템 V 공유 메모리의 형식이, 일반 스왑 아웃된 페이지의 포맷과 약간 다르기 때문에, 특별한 처리가 더 필요하기 때문이다. `swapin` 연산이 없는 경우엔, 리눅스는 이를 일반 페이지여서 특별히 처리가 필요 없다고 생각한다. 이제 비어있는 물리적 페이지를 할당하고, 스왑 아웃 되었던 페이지를 스왑 파일에서 읽어들인다. 어느 스왑 파일의 어디에 페이지가 있는지 알려주는 정보는, 해당하는 유효하지 않은 페이지 테이블 엔트리에서 얻는다.

만약 페이지 폴트를 발생한 접근이 쓰기가 아니라면, 페이지는 여전히 스왑 캐시에 남아 있으며, 메모리로 가져온 페이지 테이블은 쓰기가 안된다고 표시가 된다. 뒤에 이 페이지에 쓰기를 시도하면, 또 다른 페이지 폴트가 발생하고, 이 시점에서 그 페이지는 더티로 표시되고, 스왑 캐시에서 엔트리를 제거하게 된다. 페이지에 기록한 것이 없고 다시 스왑 아웃될 필요가 있다면, 그 페이지는 이미 스왑 파일에 있기 때문에 리눅스는 페이지를 스왑 파일에 쓸 필요가 없게 된다[22].

스왑 파일로부터 페이지를 가져오도록 한 접근이 쓰기 연산이었다면, 이 페이지는 스왑 캐시에서 제거되고, 페이지 테이블 엔트리는 더티, 쓰기 가능으로 표시된다.

번역 : 심마로, 고양우, 정직한, 김기용, 신문석, 이대현
정리 : 이호

역주 21) 앞의 `swapout` 연산과 마찬가지로 `vm_area_struct` 자료구조에 있는 `vm_ops` 포인터에(`vm_operations_struct` 구조체) 이 포인터가 들어 있다. (flyduck)

역주 22) 스왑 캐시에서 나온바와 같이 스왑 파일에 있는 내용과 메모리에 있는 내용이 달라진 경우에만 스왑 캐시에서 제거할 수 있도록, 첫번째 페이지 폴트에서는 메모리로 가져오기만 하고, 두번째 페이지 폴트가 발생할 때 스왑 캐시에서 제거하게 된다. (flyduck)

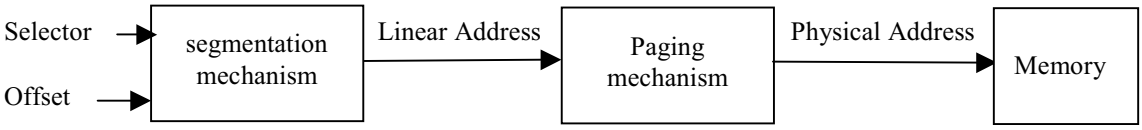
참고 자료. 인텔 386 보호모드 메모리 아키텍쳐

이호 (flyduck)

커널에서 메모리 관리 시스템의 구현은 해당 CPU의 도움을 받아야 한다. 리눅스의 메모리 관리 시스템을 이용하려면 CPU에서 페이징과 메모리 보호, 페이지 폴트 처리를 할 수 있는 메커니즘을 제공해야 하며 인텔 x86 계열의 CPU에서는 80386에서부터 이러한 메모리 아키텍쳐를 제공하고 있다. 여기서는 x86 계열의 메모리 아키텍쳐를 간단히 살펴보도록 한다.

8086 CPU는 16비트 세그먼트(segment) 레지스터와 16비트 오프셋을 중첩하여 20비트, 즉 1MB 크기의 주소공간을 제공한다. 80286에서는 8086과 똑같은 주소공간을 제공하는 실제모드(real mode)와 함께, 새로운 방식의 주소공간을 제공하는 보호모드(protected mode)가 도입되었다. 80286 보호모드에서 세그먼트 레지스터는 셀렉터(selector)라는 이름으로 바뀌었고, 셀렉터를 24비트의 베이스 주소(base address)로 바꾸어주는 테이블인 디스크립터 테이블(descriptor table)이 등장했다. 이 모드에서는 24비트의 베이스 주소와 16비트의 오프셋을 더하여 모두 24비트의 주소공간, 즉 16MB의 주소공간을 제공하였다. 여기서 셀렉터와 디스크립터 테이블을 이용하여 선형 주소공간(linear address space)의 일부를 가리킬 수 있도록 하는 것을 세그멘테이션(segmentation)이라고 한다. 80386에서는 이러한 세그멘테이션 외에 메모리 관리에 필수적인 페이징 메커니즘이 추가되고 메모리 공간도 32비트, 즉 4GB로 확장되었다.

80386에서 메모리 상의 주소를 가리키는 데에는 16비트의 셀렉터(selector) 레지스터와 32비트의 오프셋(offset)이 사용된다. 이들은 세그멘테이션 메커니즘을 거쳐 선형 주소(linear address)[23]로 변환되고, 다시 이 주소는 페이지 테이블을 이용한 페이징 메커니즘을 거쳐 물리적인 실제 주소(physical address)로 바뀌게 된다. 알파 AXP와 같은 다른 CPU에서는 세그멘테이션이라는 것을 제공하지 않으며, 이는 인텔 CPU의 특성이라고 할 수 있다. 이는 세그먼트 레지스터에서부터 시작한 잔상이라고 할 수 있으며, 리눅스 역시 이 기능을 사용하지 않고 있다. 다만 인텔 CPU에서 동작하는 다른 운영체계와 마찬가지로 세그멘테이션을 거쳐 나오는 선형 주소공간을 사용자 주소공간과 커널 주소공간으로 분리하여, 사용자 주소공간에 3GB를 커널 주소공간으로 1GB를 할당해 놓고 있다.



셀렉터는 디스크립터 테이블에 대한 인덱스와, 어떤 디스크립터 테이블을 가리키는지를 나타내는 TI (Table Indicator) 항목, 그리고 이를 사용할 수 있는 레벨을 나타내는 RPL(Requestor Privilege Level) 세가지로 이루어져 있다. TI 항목이 0이면 인덱스는 전역 디스크립터 테이블(Global Descriptor Table, GDT)에 있는 디스크립터를 가리키고, TI 항목이 1이면 지역 디스크립터 테이블(Local Descriptor Table LDT)를 나타낸다. 여기서 GDT는 커널 모드에서 사용되는 테이블이고, LDT는 사용자 모드에서 사용되는 테이블이다. 보통 GDT'는 커널 모드용으로 하나가 있으며, LDT는 각 프로세스별로 하나씩 만들어진다. 이들 테이블의 시작 위치는 각각 GDTR, LDTR이라는 레지스터가 가리키고 있다.

디스크립터 테이블은 64비트 크기로, 32비트 크기의 베이스 주소와 20비트 크기의 범위(limit), 그리고 기타 여러 항목으로 이루어져 있다. 여기서 베이스 주소는 4GB의 선형 주소공간에서의 시작 위치를 가리키고, 범위는 베이스 주소에서 시작하여 접근이 가능한 메모리 범위를 나타낸다. 이것은 20비트 크기이긴 하지만 입도 비트(granularity bit)가 설정되어 있으면 4KB 단위의 범위를 나타내므로 모두 4GB 크기의 범위를 가질 수 있다. 이렇게 나온 베이스 주소에 오프셋을 더하면 실제 선형 주소공간에서의 주소가 나오게 된다. 즉, 세그멘테

역주 23) 이 선형 주소는 커널에서 생각하는 가상 주소와 같은 것이라고 생각하면 된다. (flyduck)

이션 메커니즘에서는 셀렉터를 이용하여 디스크립터를 찾고, 여기 있는 베이스 주소에 오프셋을 더하여 선형 주소공간에서의 주소를 얻는 역할을 한다.

이렇게 얻어진 선형 주소는 실제 주소가 아니며, 페이징 메커니즘을 거쳐야 실제 주소를 얻을 수 있다. 페이징 메커니즘에서는 이 선형 주소를 다시 10비트 크기의 페이지 디렉토리 인덱스(page directory index), 10비트 크기의 페이지 테이블 인덱스(page table index), 12비트 크기의 오프셋으로 쪼갠다. 페이지 디렉토리 인덱스를 가지고 페이지 디렉토리에서 페이지 테이블의 주소를 얻을 수 있다. 다시 페이지 테이블 인덱스를 가지고 앞의 페이지 디렉토리가 가리키는 페이지 테이블에서 페이지 프레임(page frame)의 위치를 얻을 수 있다. 이렇게 얻어진 페이지 프레임 주소에 오프셋을 더하면 실제 물리적인 주소가 나오게 된다. 이는 앞의 그림 3.3에서 나오는 3단계 페이지 테이블에서 하나를 빼서 2단계 페이지 테이블을 생각하면 된다. 여기서 오프셋은 12비트이므로 하나의 페이지 프레임은 2^{12}, 즉 4KB의 크기를 가지며, 리눅스에서 정의된 페이지 크기는 이 값이다. 이렇게 페이징 메커니즘을 통하여 선형 주소는 실제 물리적인 주소로 변환되며, 리눅스는 CPU의 이런 지원을 통하여 페이징을 구현할 수 있다.

4장

프로세스 (Processes)

이 장에서는 프로세스가 무엇이며 리눅스 커널이 어떻게 프로세스를 만들고 관리하고 없애는지를 설명한다.

프로세스는 운영체제 안에서 작업을 수행한다. 프로그램은 디스크에 실행 가능한 형태로 저장되어 있는 기계어 명령과 자료의 집합인데, 이 자체는 수동적인 존재이다. 한편 프로세스는 동작중인 프로그램으로 생각할 수 있다. 즉 프로세서가 기계어 명령들을 실행함에 따라 끊임없이 변화하는 동적인 존재이다. 프로그램의 명령어와 데이터 뿐만 아니라, 프로세스는 프로그램 카운터, CPU 레지스터, 그리고 루틴 인자, 복귀 주소, 저장된 변수같은 일시적 데이터를 포함하는 프로세스 스택도 함께 가진다. 현재 실행 중인 프로그램, 즉 프로세스는 현재 마이크로프로세서 안에서 일어나는 모든 동작을 포함한다. 리눅스는 멀티프로세싱 운영체제이다. 프로세스는 각각 고유의 권한와 책임을 갖는 별개의 태스크이다. 어떤 프로세스 하나가 비정상적으로 종료했다고 해서 이것이 시스템 내의 다른 프로세스까지 죽게 하지는 않는다. 개별 프로세스는 자신의 가상 주소공간에서 실행되며, 커널이 제공하는 안전한 방법을 통하지 않고서는 다른 프로세스와 상호작용할 수 없다.

프로세스는 살아 있는 동안 많은 시스템 자원을 사용한다. 명령을 수행하기 위해서 CPU를, 명령어와 데이터를 저장하기 위해서는 물리적인 메모리를 사용한다. 파일 시스템의 파일들을 열고 사용할 수도 있고, 시스템 내의 물리적인 장치들을 직접 또는 간접적으로 사용할 수도 있다. 리눅스는 여러 시스템 자원을 관리하고 프로세스들을 공평하게 관리하기 위해서 프로세스 자신과 프로세스가 가지고 있는 시스템 자원에 대해 계속 추적하고 있어야 한다. 하나의 프로세스가 시스템의 물리적인 메모리나 CPU의 대부분을 독점한다면, 다른 프로세스들에게 공평하지 않을 것이다.

시스템에서 가장 중요한 자원은 CPU로, 대부분의 시스템에는 하나밖에 없다. 리눅스는 멀티프로세싱(multiprocessing) 운영체제인데, 그 목적은 각각의 CPU가 언제나 실행 중인 프로세스를 갖도록 하여 CPU의 활용을 극대화하는 것이다. 프로세스의 수가 CPU보다 많은 경우(대부분의 경우가 이렇다), 나머지 프로세스들은 실행되기 위해서 CPU가 사용 가능할 때까지 기다려야 한다. 멀티프로세싱이란 간단한 개념이다. 즉, 프로세스는 무언가 기달려야 하기 전까지는 (보통은 시스템 자원을 기다린다) 계속 실행되며, 기다리고 있다가 자원을 얻게 되면 프로세스는 다시 실행될 수 있다. DOS와 같은 유니프로세싱(uniprocessing) 시스템에서는 CPU는 그냥 아무것도 하지 않고 대기 시간을 낭비한다. 멀티프로세싱 시스템에서는 동시에 많은 프로세스들이 메모리 내에 존재한다. 프로세스가 무언가 기다려야 할 때마다 운영체제는 CPU를 빼앗아 다른 좀 더 적당한 프로세스가 사용하도록 한다. 어떤 프로세스가 다음에 실행될 가장 적당한 것인지 선택하는 일은 스케쥴러의 몫이고, 리눅스는 공정을 기하기 위해 여러가지의 스케쥴링 정책을 사용한다.

리눅스는 여러가지 형태의 실행 파일을 지원하는데, ELF, JAVA 등이 그 중 하나다. 이들은

프로세스가 시스템의 공유 라이브러리(shared library)를 사용할 수 있도록 하는 것과 같은 일을 위해 투명하게 관리해야 한다.

4.1 리눅스 프로세스

include/linux/sched.h 참조

리눅스가 시스템 내의 프로세스들을 관리할 수 있도록, 각각의 프로세스는 `task_struct`라는 자료구조로 표현된다 (태스크와 프로세스는 리눅스에서 같은 의미로 사용된다). `task` 벡터는 시스템에 있는 `task_struct` 구조를 가리키는 포인터들의 배열이다. 이는 시스템이 가질 수 있는 프로세스의 수가 `task` 벡터의 크기로 제한되어 있다는 것을 의미한다. 이 크기의 기본값은 512개이다. 프로세스가 만들어지면 시스템 메모리에서 새로운 `task_struct`가 할당되어 `task` 벡터에 추가된다. 현재 실행되고 있는 프로세스를 찾기 쉽게 하기 위해서, 이를 `current` 포인터가 가리키고 있다.

일반적인 프로세스 뿐 아니라 리눅스는 실시간(real time) 프로세스도 지원한다[24]. 이 프로세스들은 외부에서 발생하는 사건(event)에 매우 빨리 반응해야 하므로 (다시 말하면 실시간으로), 스케쥴러는 이들을 일반 사용자 프로세스와는 다르게 취급한다. `task_struct` 자료구조는 방대하고 복잡하지만, 내부 항목들을 여러개의 기능 영역으로 구분할 수 있다.

상태(State) 프로세스는 수행되면서 주변 상황에 따라서 상태를 변경한다. 리눅스 프로세스들은 다음과 같은 상태를 가진다[25].

- **실행중(Running)** 프로세스가 실행중이거나(현재 프로세스이거나), 언제든지 실행할 수 있는 준비가 되었음(시스템의 CPU 중 하나에 할당되는 것을 기다리고 있는 것)을 나타낸다.
- **대기중(Waiting)** 프로세스가 이벤트나 자원이 할당되길 기다리는 중임을 나타낸다. 리눅스는 두가지 종류 - 인터럽트 허용(interruptible)과 인터럽트 금지(uninterruptible) - 의 프로세스 대기상태를 가지고 있다. 인터럽트가 허용되는 대기상태의 프로세스는 시그널에 의해 인터럽트될 수 있고, 인터럽트가 금지된 대기상태의 프로세스는 하드웨어를 직접 기다리면서 어떤 환경하에서도 인터럽트되지 않는다[26].
- **중단됨(Stopped)** 프로세스가 중단된 경우로, 대개 시그널을 받았을 경우이다. 프로세스를 디버그할 때 이런 상태에 있다.
- **좀비(Zombie)** 이것은 정지된 프로세스이지만, 어떤 이유때문에 여전히 `task_struct` 자료구조를 `task` 벡터에 가지고 있는 경우이다. 용어에서 느낄 수 있듯이, 죽은 프로세스이다.

스케쥴링 정보 스케쥴러는 시스템에 있는 프로세스 중 어느 것이 가장 실행되기에 적당한지를 공정하게 판단하기 위해 이 정보를 필요로한다.

역주 24) 실시간이라는 의미는 어떤 사건이 발생하였을 때 이것이 어느 시간 이내에 처리되어야 한다는 것이다. 즉 더 중요한 사건이 발생하면 덜 중요한 일은 그만두고 이를 빠른 시간 내에 처리하는 것이다. 이를 위해 실시간 처리를 하는 운영체제(real time operating system, RTOS)는 우선순위(priority)를 사용하여, 어떤 프로세스를 수행하고 있더라도 우선순위가 더 높은 프로세스가 등장하면 하던 일을 멈추고 해당 프로세스를 수행하게 되며, 이 프로세스가 종료되거나 우선순위가 낮아지거나 더 높은 우선순위를 갖는 프로세스가 등장하지 않는 이상 계속 이 프로세스를 수행하게 된다. 이런 점에 있어서 리눅스는 실시간 프로세스가 일반 프로세스보다 먼저 수행되긴 하지만, 실시간 프로세스를 위해 프로세스를 중단하지 않고, 더 높은 우선순위의 프로세스라도 할당된 시간이 지나면 스케쥴링이 되므로 RTOS라고 할 수는 없다. (flyduck)

25) REVIEW NOTE : SWAPPING 상태는 사용되지 않는 것 같아 제외했다.

역주 26) 세마포어를 기다리거나 파일을 읽을 수 있게 되길 기다리는 것처럼 자원을 기다리는 일반적인 대기상태는 대개 인터럽트 가능한 상태이다. 인터럽트가 금지되는 대기상태는 스왑파일에서 메모리로 페이지를 읽어들이는 것과 같이 임계지역에서 일이 끝마치길 기다리고 있는 상태이다. (flyduck)

식별자(Identifier) 시스템의 모든 프로세스는 프로세스 식별자를 가지고 있다. 프로세스 식별자는 task 벡터에 대한 인덱스는 아니고, 그냥 단순한 숫자이다. 모든 프로세스는 또한 사용자 식별자와 그룹 식별자를 가지고 있는데, 이것들은 이 프로세스가 시스템에 있는 파일과 장치에 대한 접근하는 것을 제어하는 데 사용된다.

프로세스간 통신 리눅스는 전통적인 유닉스의 IPC 메커니즘인 시그널, 파이프, 세마포어와 함께, 시스템 V IPC 메커니즘인 공유 메모리, 세마포어, 메시지 큐 등을 지원한다. 리눅스에서 지원되는 IPC 메커니즘에 대해서는 5장에서 설명한다.

연결(Link) 리눅스 시스템에서, 다른 프로세스와 무관한 프로세스는 없다. 시스템의 모든 프로세스는 - 최초의 프로세스를 제외하고 - 부모 프로세스를 가진다. 새로운 프로세스는 생성되는 것이 아니라 이전의 프로세스로부터 복사(copy), 혹은 복제(clone)된다. 프로세스를 나타내는 task_struct는 모두, 부모 프로세스, 형제(sibling, 부모가 같은 프로세스들) 프로세스, 자신의 자식(child) 프로세스들에 대한 포인터를 가지고 있다. pstree 명령을 실행하여 리눅스 시스템에 실행중인 프로세스들간의 가족 관계를 볼 수 있다.

```
init(1)-+-crond(98)
        |-emacs(387)
        |-gpm(146)
        |-inetd(110)
        |-kerneld(18)
        |-kflushd(2)
        |-klogd(87)
        |-kswapd(3)
        |-login(160)---bash(192)---emacs(225)
        |-lpd(121)
        |-mingetty(161)
        |-mingetty(162)
        |-mingetty(163)
        |-mingetty(164)
        |-login(403)---bash(404)---pstree(594)
        |-sendmail(134)
        |-syslogd(78)
        `-update(166)
```

더불어, 시스템 내의 모든 프로세스들은 init 프로세스의 task_struct 자료구조에서 시작하는 이중 연결 리스트로 연결되어 있다. 이 리스트는 리눅스 커널이 시스템 내의 모든 프로세스들을 들여다볼 수 있게 한다. ps나 kill 등의 명령을 지원하려면 이렇게 할 필요가 있다.

시간과 타이머 커널은 프로세스의 생성시간과 살아있는 동안 소비하는 CPU 시간 등을 계속 추적한다. 커널은 매 클럭 틱(tick)마다, 현재 프로세스가 시스템 모드와 사용자 모드에서 사용한 시간의 양을 jiffies 단위로 갱신한다. 리눅스는 또한 간격 타이머(interval timer)도 지원하는데, 프로세스는 시스템 콜을 사용하여 타이머를 설정하고 지정한 시간이 지나면 자신에게 시그널을 보낼 수 있도록 한다. 이 타이머는 한번만 발생하는(single-shot) 타이머일 수도, 주기적으로 발생하는 타이머일 수도 있다.

파일 시스템 프로세스는 원할 때 파일을 열고 닫을 수 있으며, task_struct에는 각 열린 파일의 기술자(descriptor)에 대한 포인터와, 두개의 VFS inode 포인터를 가지고 있다. VFS inode는 각각 파일 시스템에 있는 파일이나 디렉토리를 유일하게 기술하는 것으로, 하부 파일 시스템에 대한 동일한 인터페이스를 제공하는 것이다. 리눅스가 파일 시스템을 어떻게 지원하는지는 9장에서 설명한다. 첫번째 VFS inode는 프로세스의 루트(홈 디렉토리)를 가리키고, 두번째 것은 *pwd* 디렉토리라고도 불리는 현재 디렉토리이다. *pwd*는 유닉스 명령어인 pwd에서 유래된 것으로, *print working directory*(작업 디렉토리를 출력하라)의 약자이다. 이 두 VFS inode에는 count 항목이 있어서, 몇 개의 프로세스가 그들을 참조하

고 있는지를 나타낸다. 따라서, 어떤 디렉토리나 그 디렉토리의 하위 디렉토리가 한 프로세스의 *pwd* 디렉토리로 설정되어 있다면 그 디렉토리를 삭제할 수 없다.

가상 메모리 대부분의 프로세스(커널 스레드와 데몬을 제외한)는 가상 메모리를 가지며, 리눅스 커널은 이 가상 메모리가 시스템의 실제 메모리 어디와 연결되어 있는지를 추적해야 한다.

프로세서 고유 컨텍스트(Processor Specific Context) 프로세스는 시스템의 현재 상태의 총합으로 생각할 수 있다. 프로세스는 실행될 때마다, 프로세서의 레지스터와 스택 등을 사용한다. 이것이 프로세스 컨텍스트이며, 프로세스가 중단될 때 CPU 고유의 컨텍스트들은 모두 그 프로세스의 `task_struct`에 저장되어야 한다. 스케쥴러가 이 프로세스를 다시 시작할 때, 이 컨텍스트는 이 정보로부터 복구된다.

4.2 식별자(Identifiers)

리눅스는 다른 유닉스들과 같이 시스템에 있는 파일과 이미지에 대한 접근 권한을 검사하기 위해서 사용자 식별자와 그룹 식별자를 사용한다. 리눅스 시스템의 모든 파일들은 소유권과 접근 권한을 가지며, 접근권한은 사용자들이 파일이나 디렉토리에 대한 접근 방식을 다룬다. 기본적인 권한들은 읽기, 쓰기와 실행으로 파일의 소유자, 특정 그룹에 속하는 프로세스들, 시스템의 모든 프로세스들의 세가지 종류의 사용자에 할당된다. 각각의 사용자 계층은 각기 다른 권한을 가질 수 있다. 예를 들면, 어떤 파일에 대해서 소유자는 읽기와 쓰기를 할 수 있지만, 그룹은 읽기만 할 수 있고, 시스템의 다른 프로세스들은 접근하지 못하도록 할 수 있다.

REVIEW NOTE : 추가하여 비트를 할당하는 것을 (777) 설명하라.

그룹은 리눅스에서 한명의 개별 사용자나 시스템의 모든 프로세스들이 아닌, 사용자들의 모임에 파일이나 디렉토리의 권한을 주는 방법이다. 예를 들면, 소프트웨어 프로젝트에 참가하는 사람들을 하나의 그룹으로 만들고 이 사람들만 프로젝트의 소스 코드를 읽고 쓸 수 있도록 할 수 있다. 하나의 프로세스는 여러 그룹에 속할 수 있고 (기본값은 최대 32개[27]) 이것들은 각 프로세스의 `task_struct`에 있는 `groups` 벡터에 저장되어 있다. 프로세스가 속해 있는 그룹 중의 하나가 파일에 접근 권한을 가지고 있다면, 그 프로세스는 그 파일에 대한 해당 그룹 접근 권한을 가지게 된다.

프로세스의 `task_struct`에는 네 쌍의 사용자 식별자와 그룹 식별자가 있다.

uid, gid 프로세스를 실행시킨 사용자의 사용자 식별자, 그룹 식별자

효력(effective) uid, gid 어떤 프로그램은 uid와 gid를 프로세스를 실행시킨 사용자의 것으로부터 자신의 것(실행 이미지를 기술하는 VFS inode에 저장된 속성)으로 변화시킬 수 있다. 이러한 프로그램은 *setuid* 프로그램으로 알려져 있으며, 이런 프로그램은 특히 네트워크 데몬과 같이 다른 프로세스의 한켠에서 실행되고 있는 서비스의 권한을 제한하기 위한 유용한 방법이 된다[28]. 효력 uid와 gid는 *setuid* 프로그램의 uid와 gid이며, 원래의 uid와 gid는 그대로 남는다. 커널은 특권 권한을 검사할 때 효력 uid와 gid를 검사한다.

파일 시스템 uid, gid 이것은 효력 uid, gid와 거의 같으며, 파일 시스템의 접근 권한을 검사

역주 27) 이 값은 아래나오는 `groups` 벡터 크기에 의해 제한된다. 이 값은 `NGROUPS`로 정의되어 있다. include/linux/sched.h 참조 (flyduck)

역주 28) setuid는 `passwd`같은 프로그램이 일반 사용자가 실행하였더라도 root의 권한을 획득하여 `/etc/passwd` 또는 `/etc/shadow` 파일을 수정할 수 있게도 하지만, 반대로 웹서버(httpd)같은 프로그램을 root가 실행하였더라도 nobody의 권한으로 바꾸어 다른 시스템 파일에 접근하지 못하게 하기도 한다. (flyduck)

할 때 사용된다. NFS 마운트된 파일시스템에서 사용자 모드인 NFS 서버가 파일을 접근할 때 서버로서가 아니라 특정 프로세스로서 파일을 접근해야 하기 때문에 필요하다. 이러한 경우에는 파일 시스템 uid와 gid만 변경된다 (효력 uid, gid는 변경되지 않는다). 이렇게 함으로써 악의를 가진 사용자가 NFS 서버에게 kill 시그널을 보낼 수 있게 되는 것을 막는다. kill 시그널은 특정 효력 uid와 gid를 가진 프로세스에게만 전달된다.

저장된(saved) uid와 gid 이는 POSIX 표준의 요구사항에 따른 것이며 시스템 콜을 이용하여 프로세스의 uid와 gid를 바꾸는 프로그램이 사용한다. 원래의 uid와 gid가 바뀌어 있는 동안 실제 uid와 gid를 저장하는데 사용된다.

4.3 스케쥴링(scheduling)

모든 프로세스는 어떨 때는 사용자 모드(user mode)로, 또 어떨 때는 시스템 모드(system mode)로 실행된다. 하드웨어가 이러한 모드를 지원하는 방법은 사용하는 하드웨어에 따라 다르지만, 일반적으로 사용자 모드에서 시스템 모드로 전환하거나 반대로 전환하는 안전한 메커니즘이 있다. 사용자 모드는 시스템 모드에 비하여 훨씬 적은 권한을 갖고 있다. 프로세스는 시스템 콜을 할 때마다 사용자 모드에서 시스템 모드로 전환되어 계속 실행되게 된다. 이 시점에 커널은 프로세스의 다른 한편에서 실행된다. 리눅스에서 프로세스는 현재 실행중인 프로세스를 선점하지 않는다 (non-preemptive). 즉, 자기가 실행되기 위하여 다른 프로세스를 중단시킬 수 없다[29]. 각 프로세스는 어떤 시스템 이벤트가 발생하기를 기다려야만 할 때 CPU를 내놓아야겠다고 판단한다. 예를 들어, 프로세스는 파일에서 한 글자를 읽어오기 위하여 기다려야 할 때가 있다. 이 기다림은 시스템 콜 도중에 즉, 시스템 모드에서 발생한다. 프로세스는 파일을 열고 읽기 위하여 라이브러리 함수를 사용하며, 이를 위하여 차례로 열린 파일에서 글자를 읽는 시스템 콜을 호출한다. 이 경우에 기다려야 하는 프로세스는 일시 중단이 되고 다른 실행될 만한 프로세스가 선택되어 실행된다.

프로세스는 항상 시스템 콜을 호출하며 따라서 종종 기다리게 된다. 그럼에도 불구하고 어떤 프로세스는 기다리게 될 때까지 너무 많은 CPU 시간을 사용할 수 있으며, 이러한 경우에 리눅스는 선점형 스케쥴링(pre-emptive scheduling)을 사용한다. 이 정책에서는 각각의 프로세스가 200ms 정도의 짧은 시간동안만 실행되며[30], 이 시간이 지나면 다른 프로세스가 선택되어 실행되며, 원래의 프로세스는 자신의 차례가 올 때까지 기다리게된다. 이런 작은 시간의 단위를 타임 슬라이스(time-slice)라고 한다.

kernel/sched.c schedule() 참조

실행할 수 있는 프로세스 중에서 가장 실행할만한 가치가 있는 프로세스를 골라서 실행하는 것이 스케쥴러(scheduler)의 일이다. 실행가능한 프로세스는 CPU가 자신을 실행하길 기다린다. 리눅스는 간단한 우선권에 기반한 스케쥴링 알고리즘을 사용하여, 현재 프로세스와 다른 프로세스 사이에서 실행할 놈을 고른다. 리눅스가 새로운 프로세스를 시키기로 하였다면, 현재 프로세스의 상태와 프로세스와 관련있는 레지스터들, 다른 컨텍스트를 `task_struct` 자료구조에 저장한다. 그리고 나서 실행할 새 프로세스의 상태를 복원(이것도 또한 프로세서

역주 29) 이 말은 오해를 낳을 수 있는 말이다. 이 말은 리눅스가 비선점형 스케쥴링을 한다는 것이 아니다. 뒤에 나오듯이 한 프로세스가 정해진 타임 슬라이스를 초과해서 사용하면, 그 프로세스를 중단시켜 다른 프로세스를 실행하는 선점형 스케쥴링을 한다. 여기서 선점하지 않는다는 의미는 기다려야 하는 상황이 발생하여 멈추어야 하는 경우가 발생하여 자발적으로 CPU를 내놓지 않은 이상 정해진 타임 슬라이스동안 계속 실행된다는 것이다. 또한 리눅스는 커널 모드에서는 비선점형이다. 이는 커널 코드가 재진입가능하지 않게 만들어졌기 때문이다. 일단 시스템 콜이 불리면 시스템 콜이 자발적으로 CPU를 내놓지 않은 이상 (schedule(), sleep_on(), interruptible_sleep_on() 등의 함수를 불러 스케쥴링이 일어나게 하지 않는 이상), 시스템 콜이 다른 프로세스에 의해 중단되지 않는다. (flyduck)

역주 30) 200ms는 0.2초로 CPU 입장에서 결코 짧은 시간이 아니다. 하지만 대개의 경우 프로세스가 실행되는 동안 여러 I/O에서뿐만 아니라, 스왑파일에서 페이지를 읽는 것이나, 메모리 맵된 파일을 디스크에서 메모리로 읽어들이는 것처럼 기다려야 하는 경우가 많이 발생하여 이 시간을 다 쓰는 경우는 많지 않다. (flyduck)

에 따라 다르다)하고 시스템의 제어권을 그 프로세스에게 넘겨준다. 스케쥴러가 시스템 내의 실행가능한 프로세스들에게 공정하게 CPU 시간을 할당하기 위해서 각각의 프로세스에 대한 정보를 task_struct에 유지한다.

정책(policy) 그 프로세스에 적용될 스케쥴링 정책이다. 리눅스 프로세스는 보통(normal) 프로세스와 실시간(real time) 프로세스의 두 종류로 나누어 진다. 실시간 프로세스는 다른 모든 프로세스들보다 높은 우선권을 갖고 있다. 만약, 실시간 프로세스가 실행 대기중이라면, 이 프로세스가 항상 먼저 실행된다. 실시간 프로세스는 두 종류의 policy를 가질 수 있다. 하나는 라운드 로빈(round robin)이고, 다른 하나는 FIFO(first in first out)이다. 라운드 로빈 스케쥴링에서는 실행가능만 각각의 실시간 프로세스들이 차례로 실행되고, FIFO 스케쥴링에서는 각각의 실시간 프로세스들이 실행 큐에 있는 순서에 따라서 실행되며 그 순서는 절대로 바뀌지 않는다.

우선권(priority) 이값은 스케쥴러가 프로세스에 지정한 우선순위이다. 또한 이값은 프로세스가 실행될 때, 프로세스가 실행될 수 있는 시간(jiffies 단위로)이다. 프로세스의 우선순위는 시스템 콜이나 renice 명령을 사용해서 할 수 있다.

실시간 우선권(rt_priority) 리눅스는 실시간 프로세스를 지원하며, 이것들은 시스템의 실시간이 아닌 프로세스들보다 높은 우선순위를 갖도록 스케쥴링 된다. 이 항목은 스케쥴러가 각각의 실시간 프로세스들간의 상대적인 우선순위를 지정할 수 있도록 한다. 실시간 프로세스들의 우선권는 시스템 콜을 사용해서 바꿀 수 있다.

카운터(counter) 이 값은 프로세스가 실행될 수 있는 시간(jiffies 단위로)이다. 이 값은 프로세스가 처음 실행될 때 priority 값으로 설정되며, 클럭 틱에 따라서 줄어든다.

스케쥴러는 커널안에서 여러 몇몇 경우에 작동된다. 스케쥴러는 현재 프로세스를 대기큐에 넣은 다음이나, 시스템 콜이 끝난 직후, 프로세스가 시스템 모드에서 프로세스 모드로 돌아오기 바로 전에 실행된다. 또 다른 경우는 시스템의 타이머가 현재 프로세스의 counter의 값을 0으로 설정한 경우이다. 스케쥴러는 실행될 때 다음과 같은 일들을 수행한다.

kernel/sched.c
schedule() 참조

커널 작업(kernel work) 스케쥴러는 하반부 핸들러(bottom half handler)를 실행하고, 스케쥴러 작업큐(task queue)를 처리한다. 이들 가벼운 커널 스레드들은 11장에서 자세하게 다루어진다.

현재 프로세스(current process) 현재 프로세스는 다른 프로세스가 선택되기 전에 처리되어야 한다.
현재 프로세스의 스케쥴링 정책이 라운드 로빈이면 프로세스는 실행큐로 되돌아간다.
만약 태스크가 인터럽트를 허용(INTERRUPTIBLE)하고 이전에 스케쥴된 이후에 시그널(signal)을 받았으면 실행중(RUNNING) 상태로 바뀐다.
현재 프로세스가 타임아웃되면, 이것은 실행중(RUNNING) 상태가 된다.
만약 현재 프로세스가 실행중(RUNNING) 상태이면, 그 상태가 유지된다.
프로세스들 중에서 상태가 실행중(RUNNING)이거나 인터럽트 허용(INTERRUPTIBLE)이 아닌 것들은 실행큐에서 삭제된다. 이것은 스케쥴러가 수행할 프로세스를 찾는 과정에서 이들을 제외한다는 의미이다.

프로세스 선택(process selection) 스케쥴러는 실행큐에 있는 프로세스들 중에서 수행할만한 프로세스를 찾는다. (실시간 스케쥴링 정책을 따르는) 실시간 프로세스가 있으면, 이것들이 보통의 프로세스들보다 높은 가중치를 갖는다. 보통 프로세스의 가중치는 counter의 값이지만 실시간 프로세스는 counter에 1000을 더한 값이다. 따라서, 시스템에 실행가능한 실시간 프로세스가 있으면 항상 실행가능한 보통 프로세스보다 먼저 실행된다. 주어진 타임 슬라이스를 어느 정도 소모한 (즉 counter값이 감소한) 현재 프로세스는 시스템의 같은 우선순위를 가진 다른 프로세스들보다 불리한데 이것은 당연하다. 여러개의 프로세스가 똑같은 우선순위를 갖으면, 실행큐의 보다 앞쪽에 있는 것이 선택된다. 현재 프로세스는 다시 실행큐로 되돌아간다. 많은 프로세스들이 같은 우선순위의 갖는 균형

잡힌 시스템에서는, 각 프로세스가 차례로 실행된다. 이것이 라운드 로빈 스케쥴링이다. 물론, 프로세스들이 자원을 필요로 하게 되므로, 실행 순서는 바뀌게 된다.

프로세스 교체(swap process) 가장 실행할만한 프로세스가 현재 프로세스가 아니라면, 현재 프로세스는 중단되고 새로운 프로세스가 실행되어야 한다. 프로세스는 실행중에 CPU 레지스터와, 시스템의 물리적인 메모리를 사용하게 된다. 프로세스가 루틴을 호출할 때마다 레지스터에 있는 인자들을 넘겨주며, 호출한 루틴으로 돌아오기 위한 주소 등의 값을 스택에 저장해 두기도 한다. 따라서 스케쥴러가 실행될 때는 현재 프로세스의 컨텍스트 안에서 실행되는 것이다. 특권 모드인 커널 모드에 있기는 하지만, 실행중인 것은 아직 현재 프로세스이다. 이 프로세스가 중지될 때는 프로그램 카운터(PC)와 프로세서의 레지스터 전부를 포함하여 모든 기계적인 상태가 프로세스의 `task_struct` 자료구조에 저장되어야 한다. 그리고 나면 새로운 프로세스의 모든 기계적인 상태를 로드해야 한다. 이것은 시스템 종속적인 작업으로, 어떤 CPU도 정확히 동일한 방식으로 이 일을 처리하지는 않지만, 대개는 이 작업을 위한 하드웨어적인 도움이 있다.

이 프로세스 컨텍스트 교체는 스케쥴러가 마지막으로 하는 작업이다. 따라서, 이전 프로세스의 저장된 컨텍스트는 이 프로세스가 스케쥴러의 마지막에 있는 하드웨어 컨텍스트에 대한 순간사진이다. 마찬가지로, 새로운 프로세스의 컨텍스트가 로드 되었을 때, 그것은 그 프로세스의 프로그램 카운터와 레지스터 내용을 포함하여 스케쥴러의 마지막 상태를 보여주는 순간사진일 것이다.

만약, 이전 프로세스나 새로운 현재 프로세스가 가상 메모리를 사용한다면, 시스템의 페이지 테이블 엔트리를 갱신할 필요가 있다. 물론 이 행동도 아키텍쳐에 따라 다르다. 알파 AXP와 같은 프로세서는, 변환 참조 테이블(translation look-aside table) 즉 캐시된 페이지 테이블 엔트리를 사용하므로, 이전 프로세스에 속하는 캐시된 테이블 엔트리를 지워야만 한다.

4.3.1 멀티프로세서 시스템에서의 스케쥴링

여러개의 CPU를 가진 시스템은 리눅스 세계에서 그리 흔하지 않은 것이다. 그러나 리눅스를 SMP(Symmetric Multi-Processing, 대칭형 멀티프로세싱) 운영체제로 만드려는 작업이 상당히 진척되었다. 이는 시스템내의 여러 CPU간에 작업량을 공정하게 분배하는 것이다. 공정한 분배가 가장 뚜렸이 나타는 곳은 스케쥴러이다.

멀티프로세서 시스템에서는 모든 프로세서가 바쁘게 어떤 프로세스들을 실행하고 있길 바란다. 각 프로세서는 현재 프로세스가 타임 슬라이스를 다 소모하였거나, 어떤 시스템 자원을 기다려야 할 때마다, 독립적으로 스케쥴러를 실행한다. SMP 시스템에서 맨 먼저 주목할 점은 시스템에 있는 idle 프로세스[31]가 단 하나가 아니라는 것이다. 하나의 프로세서가 있는 시스템에서는 `task` 벡터의 첫번째 태스크가 idle 프로세스이다. 반면에 SMP 시스템에서는 CPU마다 하나의 idle 프로세스가 있으며, 따라서 하나 이상의 idle CPU가 있을 수 있다. 게다가 CPU마다 하나씩의 현재 프로세스가 있으므로, SMP 시스템에서는 각 프로세서별로 현재 프로세스와 idle 프로세스를 관리하여야 한다.

SMP 시스템에서 각 프로세스의 `task_struct`에는 자신이 현재 실행되고 있는 프로세서 번호(`processor`)와 마지막으로 실행하였던 프로세서의 번호(`last_processor`)가 들어있다. 어떤 프로세스를 실행하도록 선택할 때마다 다른 CPU에서 실행하지 못할 이유는 없지만, 리눅스는 `processor_mask`를 이용하여 그 프로세스가 시스템의 특정 프로세서 또는 몇개의 프로세서에서만 실행되도록 제한할 수 있다. 만약 N비트가 설정되어 있으면 그 프로세스는 프로세서 N에서만 실행될 수 있다. 스케쥴러가 실행할 새로운 프로세스를 고를 때

역주 31) idle 프로세스는 CPU가 할 일이 아무것도 없을 때 실행하는 프로세스이다. idle 프로세스는 말 그대로 아무일도 하지 않고, CPU에서 가장 전력을 적게 소모하는 명령을 하염없이 수행한다. (flyduck)

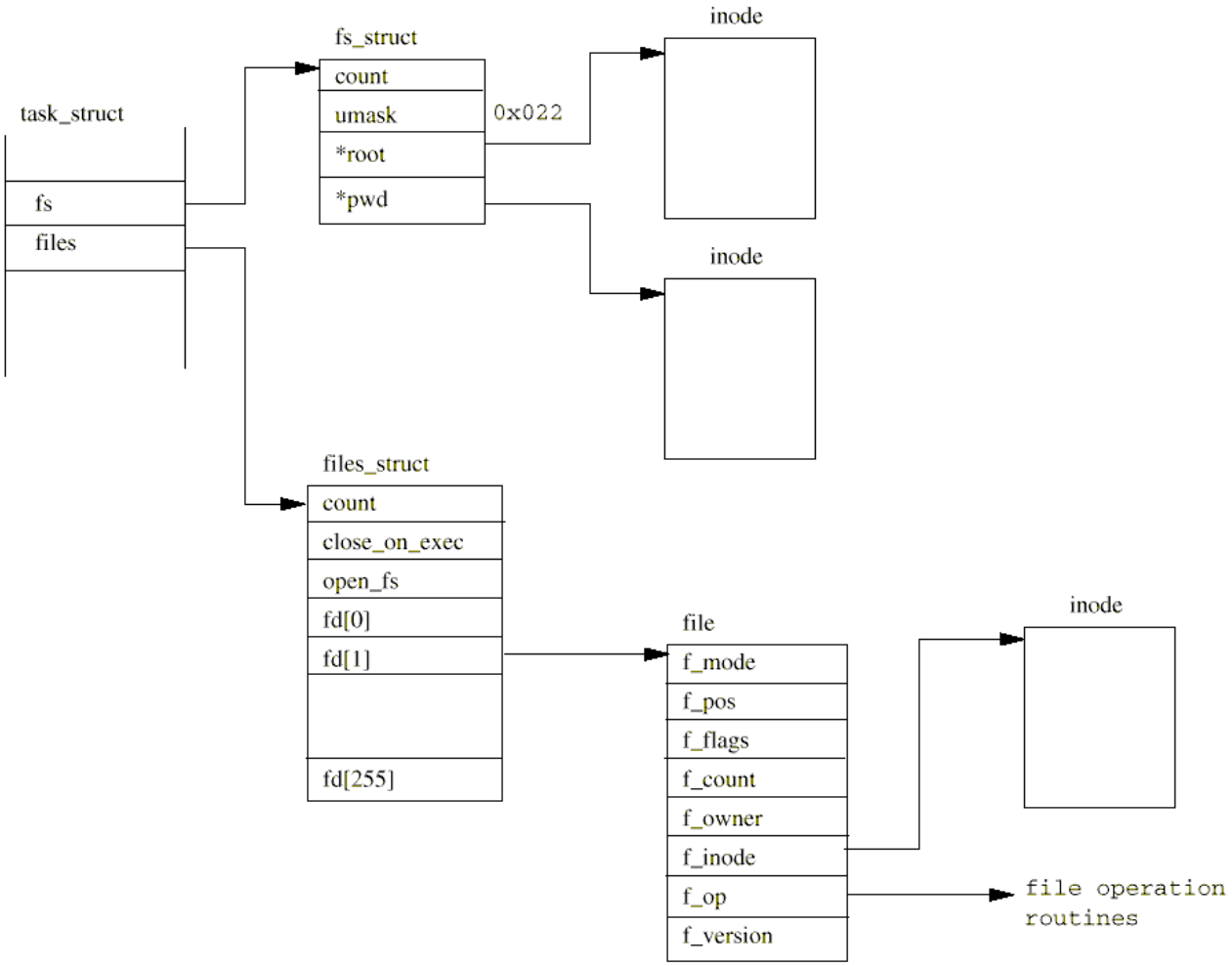


그림 4.1 : 프로세스의 파일

processor_mask에 현재 프로세서의 번호가 설정되어 있지 않은 프로세스는 고려하지 않는다. 스케쥴러는 마지막으로 현재 프로세서에서 실행되었던 프로세스에게 약간 유리하게 해준다. 이는 한 프로세스를 다른 프로세서로 옮길 때 성능상의 오버헤드가 발생하는 경우가 종종 있기 때문이다.

4.4 파일

include/linux/sched.h 참조

그림 4.1은 각 프로세스의 파일 시스템 관련 정보를 저장하는 두가지 자료구조를 보여준다. 첫번째로, fs_struct는 이 프로세스의 VFS inode에 대한 포인터와 umask를 저장하고 있다. umask는 새로운 파일이 만들어질 때의 기본 모드이며 시스템 콜에 의해 바뀔 수 있다.

두번째 자료구조인 files_struct는 현재 프로세스가 사용하고 있는 모든 파일들에 대한 정보를 가지고 있다. 프로그램은 표준 입력(standard input)에서 읽고, 표준 출력(standard output)으로 쓴다. 에러 메시지는 모두 표준 에러(standard error)로 가게 된다. 이들은 파일일 수도 있고, 단말 입/출력이나, 실제 장치일수도 있으나, 프로그램에 있어서 이들 모두는 파일로 처리된다. 각 파일은 자신을 나타내는 기술자(descriptor)를 가지며, files_struct는 이 프로세스가 사용하는 파일을 기술하는 file 자료구조에 대한 포인터를 256개까지 가진다. f_mode 항목은 파일이 만들어질 때의 모드(읽기 전용, 읽고 쓰기, 쓰기 전용)를 나타낸다. f_pos에는 다음 번에 읽거나 쓸 위치가 들어 있다. f_inode는 그 파일에 해당하는 VFS inode를 가리키고 있으며, f_ops는 그 파일에 대하여 무언가 하려고 할 때 사용할 수 있는 루틴들의 주소의 벡터를 가리킨다. 이런 함수로 데이터 쓰기 함수를 들 수 있다. 이렇

게 인터페이스를 추상화하는 것은 매우 강력하며 리눅스가 방대한 종류의 파일 유형을 지원할 수 있도록 해준다. 뒤에서 살펴보겠지만 리눅스에서 파이프는 이러한 메커니즘을 통하여 구현되었다.

하나의 파일이 열 때마다 `files_struct`에 있는 빈 `file` 포인터 중 하나가 새로운 `file` 자료구조를 가리키기 위해 사용된다. 리눅스 프로세스는 처음 시작할 때 세개의 파일 기술자가 열려 있다고 생각한다. 표준 입력, 표준 출력, 표준 에러가 그 세가지로, 이들은 대개 그 프로세스를 만든 부모 프로세스로부터 상속된다. 파일에 대한 모든 접근은 표준 시스템 콜을 통하며, 여기에 파일 기술자를 넘겨주거나 되돌려 받게 된다. 이들 기술자는 프로세스의 `fd` 벡터에 대한 인덱스 값으로, 표준 입력, 표준 출력, 표준 에러는 각각 0, 1, 2 의 기술자를 갖고 있다. 파일에 대한 접근은 `file` 자료구조의 파일 연산 루틴과 VFS inode를 같이 사용한다.

4.5 가상 메모리(Virtual Memory)

프로세스의 가상 메모리에는 여러 소스에서 나온 실행가능한 코드와 데이터가 들어 있다. 첫번째로, 로드된 프로그램의 이미지가 있다. ls 같은 명령을 예로 생각해보자. 이 명령은 다른 실행 이미지와 마찬가지로 실행가능한 코드와 데이터로 구성되어 있다. 이미지 파일에는 실행가능한 코드와 해당되는 프로그램 데이터를 프로세스의 가상 메모리에 로드하기 위해 필요한 모든 정보가 들어 있다. 두번째로, 프로세스는 처리 과정에서 필요에 의하여 - 예를 들어, 읽고 있는 파일의 내용을 담기 위하여 - (가상) 메모리를 할당받을 수 있다. 이렇게 새로 할당된 가상 메모리를 실제로 사용되기 위해서는 프로세스의 가상 메모리와 연결되어야 한다. 세번째로, 리눅스 프로세스는 파일 처리 루틴과 같이 공통적으로 유용하게 쓰이는 코드의 라이브러리를 사용하고 있다. 모든 프로세스가 똑같은 라이브러리의 복사판을 한개씩 갖고 있는다는 것은 말이 안되며, 리눅스는 실행되고 있는 여러 프로세스가 동시에 사용할 수 있는 공유 라이브러리를 사용한다. 이들 공유 라이브러리에 있는 코드와 데이터는 이 프로세스의 가상 주소 공간에 연결되어야 할 뿐만 아니라, 그 라이브러리를 공유하는 다른 모든 프로세스의 가상 주소 공간과도 연결되어야 한다.

어떤 주어진 시간 동안 한 프로세스는 가상 메모리에 들어 있는 코드와 데이터를 모두 사용하지는 않는다. 코드 중에는 어떤 특정한 경우, 예를 들어 프로세스가 시작될 때 또는 어떤 이벤트가 발생할 때에만 필요한 코드가 있다. 그리고 공유 라이브러리의 루틴도 모두 사용하는 것이 아니라 일부만 사용한다. 따라서 안 쓰일 수도 있는 코드를 실제 메모리에 모두 로드하는 것은 낭비가 될 수 있다. 이러한 낭비가 시스템내의 프로세스 수만큼 반복된다면 시스템은 매우 비효율적으로 실행될 것이다. 대신에 리눅스는 요구 페이징(demand paging)이라는 기법을 사용한다. 요구 페이징에서는 프로세스의 가상 메모리를 사용하려고 하는 순간에, 가상 메모리를 실제 메모리로 가져온다. 따라서 리눅스 커널은 프로세스의 코드와 데이터를 곧바로 실제 메모리에 로드하는 대신, 프로세스의 페이지 테이블을 수정하여 가상 영역에는 존재하고 있지만 실제로는 메모리에 있지는 않다고 표시한다. 만약 프로세스가 코드나 데이터에 접근하려고 하면, 시스템은 페이지 폴트를 발생하고, 리눅스 커널로 하여금 그 상황을 해결하라고 제어권을 넘겨준다. 이러한 페이지 폴트를 해결하려면, 리눅스는 프로세스의 주소 공간에 있는 모든 가상 메모리 영역에 대해, 그 가상 메모리가 어디에서 왔으며 어떻게 메모리에 로드할 수 있는 지를 알아야만 한다.

리눅스 커널은 이들 가상 메모리의 모든 영역을 관리할 필요가 있다. 각 프로세스의 가상 메모리의 내용은 `task_struct`에서 가리키고 있는 `mm_struct`라는 자료구조에 설명되어 있다. 프로세스의 `mm_struct` 자료구조는 로드된 실행 이미지에 대한 정보와 프로세스의 페이지 테이블에 대한 포인터도 갖고 있다. 여기에는 그 프로세스의 각 가상 메모리 영역을 나타내는 `vm_area_struct` 자료구조의 리스트에 대한 포인터도 들어 있다.

이 연결 리스트는 가상 메모리에서 오름차순으로 되어 있으며, 그림 4.2는 간단한 프로세스에서 가상 메모리의 배치상황과 그것을 관리하기 위한 커널 자료구조를 보여준다. 가상 메모리의 영역들은 여러 소스로부터 나오므로, 리눅스는 여러개의 가상 메모리 처리 루틴을

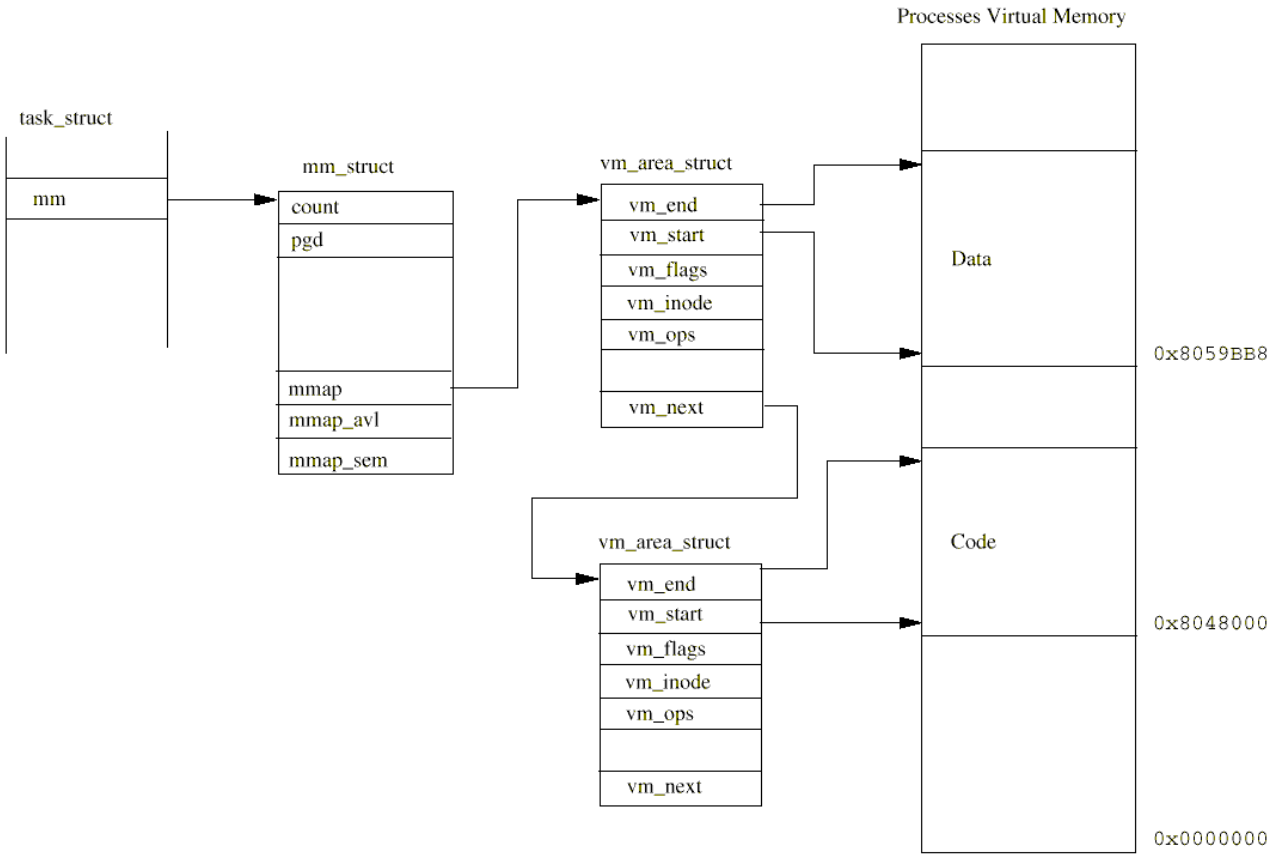


그림 4.2 : 프로세스의 가상 메모리

vm_area_struct에 있는 vm_ops를 통하여 가리키게 함으로써 인터페이스를 추상화 하였다. 이렇게 함으로써 하부 서비스가 메모리를 여러가지 다른 방식으로 관리하는 것과 상관없이 프로세스의 가상 메모리를 일관성있게 처리할 수 있게 된다. 예를 들어, 여기에는 어떤 프로세스가 메모리에 접근하는데 그 메모리가 존재하지 않을 때 불리는 루틴이 들어 있다. 페이지 폴트는 이러한 방식으로 처리된다.

리눅스 커널은 이 프로세스 용으로 가상 메모리에 새로운 영역을 만들거나, 물리적 메모리 상에 있지 않은 가상 메모리에 대한 참조를 해결할 때, 이 프로세스의 vm_area_struct 자료구조 집합을 자주 액세스하게 된다. 따라서 올바른 vm_area_struct를 찾는 데 걸리는 시간은 시스템의 성능에 큰 영향을 미친다. 이 액세스를 빠르게 하기 위하여 리눅스는 vm_area_struct 자료구조를 AVL(Adelson-Velskii and Landis) 트리의 형태로 정리해둔다. 이 트리에서는 각각의 vm_area_struct(즉, 노드)의 왼쪽 포인터와 오른쪽 포인터는 인접하는 vm_area_struct에 대한 포인터이다. 왼쪽 포인터가 가리키는 노드는 더 낮은 시작 가상 주소를 갖고 있으며, 오른쪽 포인터가 가리키는 노드는 더 높은 시작 가상 주소를 갖고 있다. 맞는 노드를 찾을 때는 트리의 루트로부터 시작하여 찾으려는 vm_area_struct를 찾을 때까지 왼쪽 또는 오른쪽 포인터를 따라간다. 물론 세상에는 공짜가 없기 때문에 새로운 vm_area_struct를 이 트리에 집어 넣는데에는 추가적인 처리 시간이 필요하다.

어떤 프로세스가 가상 메모리를 할당받을 때 리눅스는 실제 메모리를 진짜로 확보해 두지는 않는다. 대신 새로운 vm_area_struct 자료구조를 만들어 가상 메모리를 나타낸다. 이 자료구조는 프로세스의 가상 메모리 리스트에 연결된다. 프로세스가 새로운 가상 메모리 영역 안의 어떤 주소에 값을 쓰려고 하면 페이지 폴트가 발생하게 된다. 프로세서는 가상 주소를 해석하려고 하지만, 이 메모리에 대해서 페이지 테이블 엔트리가 존재하지 않기 때문에, 프로세서는 이를 포기하고 페이지 폴트 예외를 발생하며, 리눅스 커널이 이를 수정하도록 한다. 리눅스는 참조된 가상 주소가 현재 프로세스의 가상 주소 공간에 있는지 찾는다. 그렇다면 리눅스는 해당하는 PTE를 생성하고, 물리적 메모리 페이지를 할당한다. 코드나 데이터는

파일시스템이나 스왑 디스크로부터 물리적 페이지로 가져와야 할 수도 있다. 이제 프로세스는 페이지 폴트를 발생한 명령에서부터 다시 시작할 수 있으며, 이제 메모리가 물리적으로 존재하므로 작업을 계속할 수 있다.

4.6 프로세스 생성하기

시스템이 처음 시작될 때 시스템은 커널 모드에 있으며, 초기 프로세스라는 단 하나의 프로세스만 존재한다. 다른 프로세스들과 같이 초기 프로세스는 스택과 레지스터 등으로 대표되는 기계 상태를 갖고 있다. 이것들은 시스템의 다른 프로세스들이 만들어지고 실행될 때, 초기 프로세스의 task_struct 구조에 저장된다. 시스템 초기화의 마지막 단계에서, 초기 프로세스는 init라고 하는 커널 쓰레드를 시작하고 아무일도 하지 않는 루프로 들어간다. 언제나 다른 할 일이 없으면 스케쥴러는 이 idle 프로세스를 실행한다. idle 프로세스의 task_struct는 유일하게 동적으로 할당된 것이 아니고 커널이 생성될 때 정적으로 정의된 것으로, 조금 혼란스럽겠지만 init_task라고 한다.

init 커널 쓰레드(또는 프로세스)는 시스템의 첫번째 진짜 프로세스로, 프로세스 식별자로 1을 갖는다. 이 프로세스는 시스템 초기화의 일부를 담당하고 (시스템 콘솔을 열고, 루트 파일 시스템을 마운트하는 것 등), 시스템 초기화 프로그램을 실행한다. 시스템에 따라서 다르지만 /etc/init, /bin/init, /sbin/init 중의 하나이다. init 프로그램은 시스템에서 새 프로세스들을 만들기 위해서 /etc/inittab이라는 스크립트 파일을 사용한다. 이 새 프로세스들은 또 다른 프로세스들을 만들기도 한다. 예를 들면, getty 프로세스는 사용자가 로그인을 시도할 때 login 프로세스를 만들기도 한다. 시스템내의 모든 프로세스들은 init 커널 쓰레드의 자손이다.

kernel/fork.c
do_fork() 참조

새 프로세스들은 예전의 프로세스들을 복제하거나 현재의 프로세스를 복제하면서 생성된다. 새 태스크는 시스템 콜(*fork*나 *clone*)에 의해서 만들어 지며, 복제는 커널이 커널 모드에서 한다. 시스템 콜의 마지막에는 스케쥴러가 자신을 선택하여 실행하길 기다리는 새로운 프로세스가 있게 된다. 새 task_struct 자료구조가 시스템의 실제 메모리에서 할당되고, 하나 또는 몇 개의 페이지가 복제된 프로세스의 스택(사용자와 커널) 용으로 할당된다. 시스템에 있는 식별자들 중에서 유일한 새로운 식별자가 만들어진다. 그리고 복제된 프로세스는 당연하게도 부모 프로세스의 식별자를 가지고 있다. 새 task_struct가 task 벡터에 할당되고, 예전 (current) 프로세스의 task_struct의 내용이 복제된 task_struct에 복사된다.

프로세스를 복제할 때, 리눅스는 두 프로세스가 별도의 복사본을 사용하는게 아니라 자원을 공유하도록 한다. 프로세스의 파일들, 시그널 핸들러와 가상 메모리가 여기에 해당된다. 자원을 공유할 때 이들의 count 값을 증가시켜 두개의 프로세스 모두가 자원 사용을 마치기 전에는 할당을 해제하지 못하도록 한다. 그래서, 예를 들어 복제된 프로세스와 가상 메모리를 공유할 때, 이 프로세스의 task_struct는 원래 프로세스의 mm_struct에 대한 포인터를 갖고, mm_struct의 count 값은 증가되어서 이를 공유하고 있는 프로세스의 개수를 나타낸다.

프로세스의 가상메모리를 복제하는 데에는 좀 더 트릭을 사용한다. 새 vm_area_struct 자료구조들은 이들을 포함하는 mm_struct 자료구조와 복제된 프로세스의 페이지 테이블과 함께 만들어 져야 한다. 프로세스의 가상 메모리는 이 시점까지는 전혀 복사되지 않는다. 가상 메모리의 일부는 실제 메모리에 있고, 또 다른 부분은 현재 실행중인 프로세스의 실행 이미지에 있으며, 어떤 부분은 스왑 파일에 있을 수 있으므로, 이것은 상당히 어렵고 시간을 소요하는 일이다. 대신에 리눅스는 "기록시 복사(copy on write)"라는 기술을 사용하는데, 이것은 두 프로세스 중 하나가 기록을 시도할 때만 가상 메모리를 복사하는 것이다. 가상 메모리 중에서 기록되지 않은 부분은 (설사 그것이 쓸 수 있는 영역이라고 하더라도) 아무 문제없이 두 프로세스 사이에서 공유된다. 실행 코드와 같은 읽기 전용 메모리는 항상 공유된다. "기록시 복사"가 동작하기 위해서, 쓸 수 있는 영역들의 페이지 테이블 엔트리는 읽기 전용으로 표시되고, 이를 나타내는 vm_area_struct 자료구조에는 "기록시 복사"라고 표시한다. 그러면 프로세스 중 하나가 이 가상 메모리에 쓰려고 하면 페이지 폴트가 발생한다. 이 때

리눅스는 메모리의 복사본을 만들고 두 프로세스의 페이지 목록과 가상 메모리 구조를 조정한다.

4.7 시간과 타이머

커널은 각 프로세스의 생성 시간과, 프로세스가 사용한 CPU 시간을 관리한다. 각 클럭 틱마다 커널은 현재 프로세스가 시스템 모드와 사용자 모드에서 사용한 시간의 양을 `jiffies` 단위로 계산하여 갱신한다.

kernel/itimer.c 참조

이들 요금계산용 타이머에 외에도, 리눅스는 프로세스가 지정하여 사용할 수 있는 간격 타이머를 지원한다. 프로세스는 이들 타이머를 이떤 시간이 지났을 때 자신에서 여러가지 시그널을 보내는데 사용할 수 있다. 리눅스는 세가지 종류의 간격 타이머를 지원한다.

실제(Real) 실제 시간으로서의 타이머 틱으로, 타이머가 만료되면 프로세스는 `SIGALRM` 시그널을 받는다.

가상(Virtual) 프로세스가 수행한 시간으로서의 타이머 틱으로, 만료되면 `SIGVTALRM` 시그널을 받는다.

일람(Profile) 프로세스가 수행한 시간과 프로세스의 다른 한편에서 시스템이 수행한 시간을 합친 타이머 틱으로, 만료되면 `SIGPROF` 시그널을 받는다.

kernel/sched.c do_it_virtual() 참조

kernel/sched.c do_it_prof() 참조

kernel/itimer.c it_real_fn() 참조

하나 또는 모든 간격 타이머가 실행될 수 있으며, 리눅스는 프로세스의 `task_struct` 자료구조에 필요한 모든 정보를 간직한다. 시스템 콜을 사용하여 이들 간격 타이머를 설정하고, 시작하고, 멈추고, 현재 값을 읽을 수 있다. 가상 타이머와 일람 타이머는 똑같은 방법으로 처리된다. 각 클럭 틱마다 현재 프로세스의 간격 타이머는 감소하며, 만료되면 해당하는 시그널을 받는다.

실제 시간 간격 타이머는 다른 타이머들과는 약간 다르며, 리눅스는 이들을 위해 11장에서 설명하고 있는 타이머 메커니즘을 사용한다. 각 프로세스는 자신의 `timer_list` 자료구조를 가지고 있으며, 실제 간격 타이머가 실행되고 있으면, 이를 시스템 타이머 리스트 큐에 넣는다. 타이머가 만료되면 타이머 하반부 핸들러는 이를 큐에서 제거하고 간격 타이머 핸들러를 부른다. 이 핸들러는 `SIGALRM` 시그널을 발생하고, 새로 간격 타이머를 시작하여 이를 다시 시스템 타이머 큐에 넣는다.

4.8 프로그램 실행하기

유닉스와 마찬가지로 리눅스에서는 프로그램과 명령어들은 보통 명령어 해석기(command interpreter)에 의해 수행된다. 명령어 해석기는 다른 프로세스처럼 사용자 프로세스이며, 쉘(shell)[32]이라고 불린다. 리눅스에는 여러가지 쉘이 있는데 가장 대중적인 것으로는 `sh`, `bash`, `tcsh`가 있다. `cd`나 `pwd`같이 적은 수의 내부에 직접 구현된 명령어들을 제외하고, 명령어들은 실행할 수 있는 이진 파일이다. 명령어가 입력되면 쉘은 환경변수 PATH에 저장된 프로세스의 찾기 경로(search path)에서 같은 이름을 가진 실행 이미지를 찾는다. 파일을 찾으면 이를 로드하고 실행한다. 쉘은 앞에서 설명한 *fork* 메커니즘을 이용하여 자기자신을 복제한 후, 이렇게 만들어진 새로 만들어진 자식 프로세스는 이전에 실행하고 있던 이진 이미지를 (여기서는 쉘) 찾은 파일의 실행 이미지로 교체한다. 보통 쉘은 명령이 완료되길, 즉 자식 프로세스가 종료되기를 기다린다. 여기서 쉘이 이 자식 프로세스를 백그라운드로 돌려 실행되게 할 수 있는데, 먼저 `control-Z`를 눌러서 자식 프로세스에게 `SIGSTOP` 시그널을 보내 멈추게 한다. 그리고 쉘 명령어인 `bg`를 사용하면 이를 백그라운드로 돌리고 `SIGCONT` 시그

32) 땅콩을 생각해보면, 커널은 가운데 먹을 수 있는 부분이고, 쉘은 이를 둘러 싸고 있는 것으로 인터페이스를 제공한다.

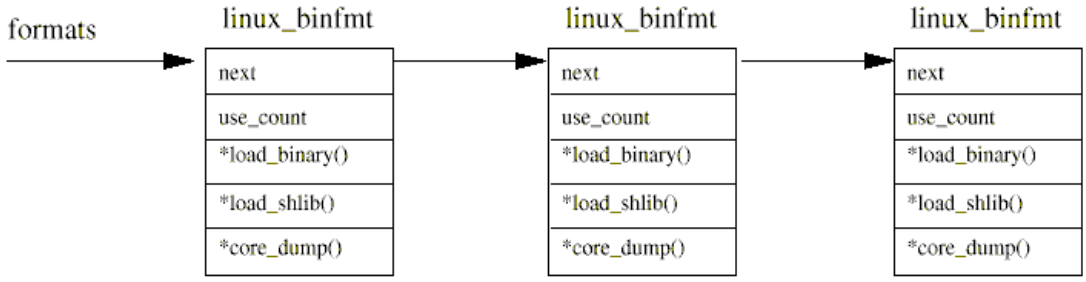


그림 4.3 : 등록된 이진 포맷들

널을 보내 다시 시작하게 한다. 이 프로세스는 종료하거나 터미널 입출력이 필요할 때까지 그대로 남아 있을 것이다.

실행 파일은 여러가지 포맷으로 되어 있을 수 있으며, 심지어 스크립트 파일도 가능하다. 스크립트 파일로 인식했다면, 이를 처리할 수 있는 올바른 해석기를 실행해야 한다. 예를 들어 `/bin/sh`는 쉘 스크립트를 해석한다. 실행할 수 있는 오브젝트 파일은 실행 코드와 데이터와 함께, 운영체계가 이를 메모리에 올리고 실행할 수 있도록 하는데 필요한 정보를 가지고 있다. 리눅스에서 가장 일반적으로 사용하는 파일 포맷은 ELF이지만, 리눅스는 어떤 오브젝트 파일 포맷도 다룰 수 있을만큼 유연하게 되어 있다.

파일 시스템처럼 리눅스는 커널을 컴파일할 때 이진 포맷을 지원하는 것을 커널에 포함할 수도 있고 모듈로 로드할 수도 있다. 커널은 지원하는 이진 포맷의 목록을 관리하고 있다가 (그림 4.3참조), 파일을 실행하려고 하면 동작하는 것을 찾을 때까지 하나씩 각 이진 포맷을 시도해본다. 일반적으로 리눅스에서 지원하는 이진 포맷은 `a.out`과 `ELF`이다. 파일을 실행할 때 파일을 모두 다 메모리로 읽어들일 필요는 없으며, 요구시 로딩(demand loading) 기술을 사용하여, 프로세스가 실행 이미지의 각 부분을 사용할 때 이것을 메모리로 가져온다. 이미지에서 안쓰이는 부분은 메모리에서 폐기된다.

fs/exec.c
do_execve()
참조

4.8.1 ELF

ELF (실행가능하고 링크할 수 있는 포맷 : Executable and Linkable Format) 오브젝트 파일 포맷은 유닉스 시스템 연구소(Unix System Laboratories)에서 디자인한 것으로, 이제는 리눅스에서 가장 일반적으로 사용하는 포맷이 되었다. `ECOFF`나 `a.out`같은 다른 오브젝트 파일 포맷과 비교하면 약간의 성능상의 오버헤드가 있지만, ELF는 좀 더 유연하다. ELF 실행 파일은 텍스트(text)라고 부르는 실행 코드와 데이터(data)를 가지고 있다. 실행 이미지 안에 있는 테이블은 어떻게 프로그램이 프로세스의 가상 메모리에 들어가야 하는지를 기술한다. 정적으로 링크된 이미지는 링커(ld)나 링크 편집기(link editor)같은 것을 이용하여, 하나의 이미지에 실행하는데 필요한 모든 코드와 데이터를 가지고 있다. 이와 함께 이미지는 자신의 메모리에서의 배치도와 처음 수행할 코드의 이미지 내의 주소를 지정하고 있다.

그림 4.4는 정적으로 링크된 ELF 실행 이미지의 배치도를 보여준다. 이것은 "hello world"를 출력하고 종료하는 간단한 C 프로그램이다. 헤더는 이것이 두개의 물리적 헤더(`e_phnum`이 2이다)가 이미지 파일의 처음을 기준으로 52바이트(`e_phoff`)에 위치하는 ELF 이미지라는 것을 이야기한다. 첫번째 물리적 헤더는 이미지에서 실행 코드를 기술한다. 이는 가상 주소 *0x8048000*에서 시작하고 65532 바이트를 갖는다. 이렇게 큰 이유는 이것이 정적으로 링크된 이미지여서, "hello world"를 출력하는 `printf()` 함수에 대한 라이브러리 코드를 모두 가지고 있기 때문이다. 이미지의 진입점(entry point), 즉 프로그램에서 처음 실행하는 명령은 이미지의 시작주소가 아니라 가상 주소 *0x8048090* (`e_entry`)이다. 이 코드는 두번째 물리적 헤더를 로드한 직후에 바로 시작된다. 이 두번째 물리적 헤더는 프로그램에서의 데이터를 나타내고, 가상 메모리의 *0x8059BB8* 위치에 로드된다. 이 데이터는 읽거나 쓸 수 있다. 여기서 파일에서 데이터의 크기는 2200바이트(`p_filesz`)인데 반해, 메모리에서의 크기는 4248바이

include/linux/
elf.h 참조

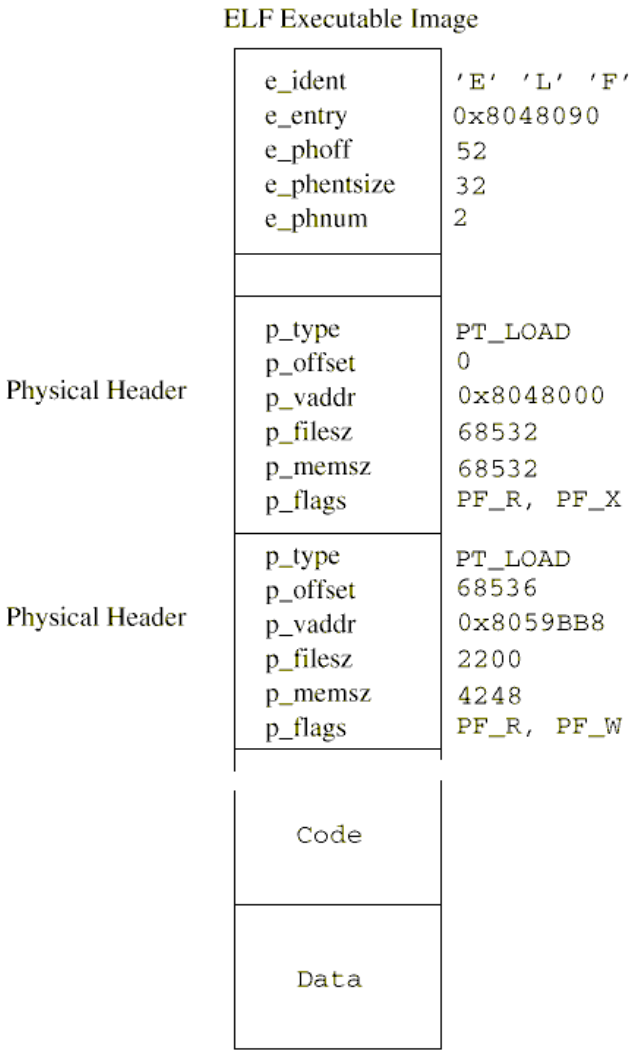


그림 4.4 : ELF 실행 파일 포맷

트인 것을 눈치챈 사람도 있을 것이다. 이는 처음 2200바이트는 미리 초기화된 데이터를 가지고 있지만, 다음에 있는 2048바이트는 실행 코드가 초기화할 데이터를 가지고 있기 때문이다.

fs/binfmt_elf.c
do_load_elf_
binary() 참조

리눅스가 프로세스의 가상 주소 공간에 ELF 실행 이미지를 로드할 때, 실제로 이미지를 올리는 것은 아니다. 리눅스는 단지 가상 메모리 자료구조인 프로세스의 `vm_area_struct` 트리와 여기에 속한 페이지 테이블들을 셋업하는 것이다. 그리고 프로그램이 실행되면서 페이지 폴트가 발생하면 프로그램의 코드와 데이터를 물리적인 메모리로 가져온다. 프로그램에서 안쓰이는 부분은 절대 메모리에 로드되지 않는다. ELF 이진 포맷 로더는 자신이 실행할 이미지가 ELF 실행 이미지가 맞다는 것을 확인하면, 프로세스의 가상 메모리에서 현재 실행 이미지를 쫓아낸다. 이 프로세스는 복제된 이미지이므로 (모든 프로세스가 마찬가지지만), 이 옛날 이미지는 부모 프로세스가 실행했던 프로그램 - 예를 들어 `bash`같은 명령어 해석 쉘 - 일 것이다. 이렇게 옛날 실행 이미지를 쫓아내는 것은 옛날 가상 메모리 자료구조를 없애고 프로세스의 페이지 테이블들을 리셋한다. 또한 설정되어 있는 모든 시그널 핸들러를 지우고, 열려진 파일들을 모두 닫는다. 이 쫓아내기 과정이 끝나면 프로세스는 새로운 실행 이미지를 받아들일 준비가 된다. 실행 이미지가 어떤 포맷이냐에 관계없이 프로세스의 `mm_struct`는 똑같은 정보로 셋업이 된다. 여기에는 이미지의 코드와 데이터의 시작과 끝을 나타내는 포인터가 있다. 이 값들은 ELF 실행 이미지 물리적 헤더를 읽는 중에 발견하게 되고, 이 헤더에서 기술하는 프로그램 섹션들은 프로세스의 가상 주소 공간에 맵핑이 된다. 이는 `vm_area_struct` 자료구조를 셋업하고 프로세스의 페이지 테이블들을 수정

할 때도 마찬가지다. mm_struct 자료구조 또한 프로그램에 전달될 인자들에 대한 포인터와 프로세스의 환경 변수에 대한 포인터도 가지고 있다.

ELF 공유 라이브러리

한편, 동적으로 링크되는 이미지는 실행하는데 필요한 모든 코드와 데이터를 가지고 있진 않는다. 이들 중 일부는 실행시에 이미지와 링크되는 공유 라이브러리(shared library)에 들어 있다. ELF 공유 라이브러리의 테이블들은 실행시에 동적 링커가 공유 라이브러리를 이미지와 연결할 때 사용한다. 리눅스는 여러개의 동적 링커를 사용한다. ld.so.1, libc.so.1, ld-linux.so.1. 이들 모두는 /lib에서 찾을 수 있다. 이 라이브러리는 언어 서브루틴 같이 공통으로 사용하는 코드를 가진다. 동적 링크를 사용하지 않는다면 모든 프로그램은 이들 라이브러의 복사본을 가지고 있어야 할 것이며, 훨씬 많은 디스크 공간과 가상 메모리를 필요로 할 것이다. 동적 링크에서 정보들은 ELF 이미지에 있는 참조하는 모든 라이브러리 함수들의 테이블에 들어 있다. 이 정보는 동적 링커에게 어떻게 라이브러리 루틴을 위치시키고 프로그램의 주소 공간에 링크시킬지를 알려준다.

REVIEW NOTE : 실행 예제를 가지고 이를 더 자세히 설명할 필요가 있는가?

4.8.2 스크립트 파일(Script File)

스크립트 파일은 실행하는데 인터프리터(interpreter)를 필요로하는 실행파일이다. 리눅스에는 아주 다양한 인터프리터가 있다. 예를 들어 wish, perl이나 tcsh같은 명령쉘이 모두 인터프리터이다. 리눅스는 인터프리터의 이름을 스크립트 파일의 첫번째 줄에 가지고 있는 표준 유닉스 표기법을 따른다. 따라서, 전형적인 스크립트 파일은 다음과 같이 시작한다.

```
#!/usr/bin/wish
```

fs/binfmt_script.c
do_load_script()
참조

스크립트 이진 로더는 이 스크립트를 처리할 인터프리터를 찾으려고 한다. 이것은 스크립트의 첫번째 줄에서 말한 실행파일을 열려고 하는 것이다. 만약 이를 열 수 있다면, 이 프로그램의 VFS inode에 대한 포인터를 가지고 스크립트 파일 해석을 시작할 수 있을 것이다. 스크립트 파일의 이름은 인자 0번(프로그램에 전달되는 첫번째 인자)에 설정되고, 다른 모든 인자들도 한 칸씩 이동하게 된다 (원래 첫번째 인자였던 것이 두번째 인자가 되는 식이다). 인터프리터를 로드하는 것은 리눅스가 모든 실행 파일을 로드하는 것과 같은 방법으로 한다. 리눅스는 각 이진 포맷을 차례로 시도하여 동작하는 것을 찾는다. 이는 이론적으로 여러개의 인터프리터와 이진 포맷들을 쌓아 올릴 수 있게 하며, 리눅스 이진 포맷 핸들러를 매우 유연한 소프트웨어로 만든다.

번역 : 윤경일, 고양우, 서창배, 이호, 정직한, 김기용,
정리 : 이호

5장

프로세스간 통신 메커니즘 (Interprocess Communication Mechanism)

프로세스들은 상호간의 활동을 조정하기 위해서 프로세스간, 그리고 커널과 통신을 한다. 리눅스는 여러 종류의 프로세스간 통신 기능(Inter-Process Commuication, IPC)을 제공한다. 리눅스는 시그널과 파이프 이외에도 시스템 V IPC를 제공하는데 시스템 V IPC는 이 기능이 처음으로 등장한 유닉스 버전의 이름을 따서 지어진 이름이다.

5.1 시그널(Signal)

시그널은 유닉스 시스템에서 프로세스간 통신을 하는 가장 오래된 방법 중의 하나이다. 이들은 하나 이상의 프로세스들에게 비동기적인 이벤트를 알리기 위해 사용된다. 시그널은 키보드 인터럽트로부터 발생되기도 하고, 프로세스가 존재하지 않는 가상 메모리 영역을 사용하려 하는 경우같은 에러 상황에서도 발생한다. 시그널은 쉘이 자식 프로세스에게 작업 관리 명령을 보낼 때에도 사용된다.

커널이나 해당하는 권한을 가지고 있는 시스템의 다른 프로세스들이 발생할 수 있는 일련의 정의된 시그널들이 있다. 이러한 시그널들을 보려면 kill 명령을 사용하면 되는데(kill -l), 필자의 인텔 리눅스 기계에는 다음과 같은 시그널들이 있다.

```
 1) SIGHUP      2) SIGINT     3) SIGQUIT    4) SIGILL
 5) SIGTRAP     6) SIGIOT     7) SIGBUS     8) SIGFPE
 9) SIGKILL    10) SIGUSR1   11) SIGSEGV   12) SIGFEP
13) SIGPIPE    14) SIGALRM   15) SIGTERM   17) SIGCHLD
18) SIGCONT    19) SIGSTOP   20) SIGTSTP   21) SIGTTIN
22) SIGTTOU    23) SIGURG    24) SIGXCPU   25) SIGXFSZ
26) SIGVTALRM  27) SIGPROF   28) SIGWINCH  29) SIGIO
30) SIGPWR
```

시그널의 개수는 알파 AXP 리눅스 시스템과 다를 수 있다. 프로세스들은 대부분의 시그널들을 무시하려면 무시할 수 있지만, 여기에는 두 개의 중요한 예외가 있다: 프로세스의 실행을 중단시키는 SIGSTOP 시그널과 프로세스를 끝내게 하는 SIGKILL 시그널은 무시할 수 없다. 그렇긴 하지만, 프로세스는 여러가지의 시그널을 어떻게 처리할 지 결정할 수 있다. 프로세스는 시그널을 블럭할 수 있고, 블럭하지 않는 경우에는 스스로 처리하거나 커널이 처리하도록 하는 것 중에 선택할 수 있다. 만약 커널에게 처리를 맡기는 경우에는 시그널에 해당하는 기본 동작이 취해지게 된다. 예를 들어서, 프로세스가 SIGFPE(부동 소수점 연산 예외) 시그널을 받은 경우의 기본 동작은 코어 덤프(core dump)를 하고 프로세스를 끝내는 것으로 되어 있다. 시그널에는 본래 우선순위가 없다. 한 프로세스에게 동시에 두 개의 시그

널이 발생하는 경우, 이 시그널들이 프로세스에 전달되는 순서나 처리되는 순서는 정해져 있지 않다. 또한 동시에 같은 시그널이 여러번 발생하는 것을 처리할 수 있는 메커니즘도 없다. 따라서, 프로세스가 SINGCONT 시그널을 한번을 받든 42번을 받든 이를 구별할 방법이 없다.

include/linux/sched.h 참조

리눅스는 프로세스의 task_struct에 저장된 정보를 사용해서 시그널 기능을 구현한다. 지원할 수 있는 시그널의 갯수는 프로세서의 워드(word) 크기에 제한을 받는다. 32비트 워드를 사용하는 시스템에서는 최대한 32개의 시그널을 지원할 수 있고, 알파 AXP와 같이 64비트 프로세서를 사용하는 경우에는 최대 64개의 시그널을 지원할 수 있다. 현재 처리 대기중인 시그널들은 signal 항목에 저장되며, 블럭된 시그널들의 마스크는 blocked 항목에 담기게 된다. SIGSTOP과 SIGKILL을 제외한 다른 모든 시그널들은 블럭킹 할 수 있다. 블럭된 시그널이 발생할 경우 그 시그널은 블럭킹을 해제할 때까지 대기 상태로 남아 있게 된다. 리눅스는 또한 발생할 수 있는 모든 시그널들을 프로세스가 어떻게 처리하는가에 대한 정보를 가지고 있는데, 이 정보는 프로세스의 task_struct에 있는 sigaction 자료구조의 배열에 저장된다. sigaction에는 여러가지 다른 정보들과 함께, 시그널 핸들러의 주소, 또는 프로세스가 해당 시그널을 무시할 것인지 혹은 커널이 그 시그널을 대신 처리하게 할 것인지를 나타내는 플래그가 들어 있다. 프로세스는 시스템 콜을 통해서 기본 시그널 핸들러를 바꿀 수 있으며, 이 시스템 콜은 해당 시그널의 sigaction과 blocked 마스크를 변경한다.

시스템 내의 프로세스들이 모두 다른 프로세스로 시그널을 보낼 수 있는 것은 아니다. 커널과 관리자는 모든 프로세스에게 보낼 수 있지만, 일반 프로세스는 같은 uid와 gid를 갖는 프로세스, 또는 같은 프로세스 그룹[33] 내의 프로세스에게만 시그널을 보낼 수 있다. 시그널은 task_struct내 signal 항목의 해당하는 비트를 설정하여 발생된다. 프로세스가 그 시그널을 블럭하지 않았고, 인터럽트 가능한 상태에서(즉 INTERRUPTIBLE 상태에서) 대기중에 있다면, 프로세스는 현재 상태를 실행중(RUNNING)으로 바꾸고 자신을 실행큐에 넣음으로써 깨어나게 된다. 이런 방법으로 시스템이 다음번 스케쥴링을 수행할때, 스케쥴러가 그 프로세스를 실행할 후보로 생각하게 된다. 기본 동작으로의 시그널 처리만이 필요하다면 리눅스는 시그널 처리을 최적화 시킬 수 있다. 예를 들어 SIGWINCH(X 윈도우가 포커스를 변경)가 발생하였고 기본 핸들러를 사용할 것이라면, 프로세스가 따로 수행할 일은 없게 되는 것이다.

시그널은 발생하는 순간 바로 프로세스로 전달되는 것이 아니라 그 프로세스가 다시 수행될 때까지 기다려야 한다. 즉 프로세스가 시스템 콜을 마치고 돌아올 때마다 signal과 blocked가 매번 검사되는데, 이때 블럭되지 않은 시그널이 존재하는 경우 비로서 프로세스로 전달되는 것이다. 이 방식은 상당히 신뢰성이 낮은 방법처럼 보이지만, 시스템 내의 프로세스들은 무슨 목적에서든(예를 들면 터미널에 문자를 찍기 위해서) 실행 시간 대부분에 걸쳐 시스템 콜을 계속 수행하므로 그렇지는 않다. 원한다면 프로세스는 시그널 발생을 기다리는 것을 선택할 수 있는데, 이 경우 인터럽트 허용 상태에서 시그널이 전달되어 올 때까지 프로세스는 멈춰 서있게 된다. 리눅스 시그널 처리 코드는 현재 블럭되지 않은 시그널에 대해서 sigaction 자료구조를 참조한다.

시그널 핸들러가 기본 핸들러로 되어 있으면 커널이 그 처리를 대신 수행하게 된다. SIGSTOP 시그널에 대한 기본 핸들러는 현재 프로세스의 상태를 중지됨(STOPPED)으로 바꾸고, 새로 실행할 프로세스를 선택하기 위해 스케쥴러를 실행한다. SIGFPE 시그널을 받으면 커널은 현재 프로세스를 코어 덤프하고 프로세스를 종료한다. 이와 달리 프로세스가 직접 자신의 시그널 핸들러를 지정했을 수도 있다. 이것은 시그널이 발생할 때마다 호출되는 것으로, sigaction 자료구조가 이 루틴의 주소를 가지고 있다. 이제 커널은 반드시 프로세스의 시그널 핸들러를 호출해야 하는데, 이것이 어떻게 이루어지는가는 프로세서에 따라 다르지만, 한가지 사실, 즉 현재 프로세스는 커널 모드에서 실행중이며 곧 사용자 모드에서 커널 혹은 시스템 루틴을 부른 프로세스로 돌아가려고 한다는 점만은 모든 CPU들이 염두에

33) REVIEW NOTE : 프로세스 그룹을 설명할 것

두고 대처하여야 하는 문제이다[34]. 이 문제는 프로세스의 스택과 레지스터를 조작함으로써 해결가능하다. 프로세스의 프로그램 카운터를 그 시그널 처리 루틴으로 설정하고, 핸들러로 전달할 인자를 스택 프레임에 추가하거나 레지스터에 담아 보내는 것이다. 이후 프로세스가 실행을 재개하면 시그널 처리 루틴은 마치 정상적인 방법으로 호출되었던 것같이 보이게 된다.

리눅스는 POSIX 호환이므로, 프로세스는 특정 시그널 처리 루틴이 호출되었을 때 어떤 시그널을 블럭할 것인지를 지정할 있다. 이것은 프로세스 시그널 핸들러가 불리는 동안 `blocked` 마스크의 값을 바꾸게 됨을 뜻한다. `blocked` 마스크는 시그널 처리 루틴이 종료될 때 원래 값으로 돌려 놓아야 한다. 그래서 리눅스는 정리용 루틴을 하나 더 불러서, 시그널을 받은 프로세스의 콜 스택에 저장해놓은 원래의 `blocked` 마스크 값을 꺼내어 복구하도록 한다. 또한 여러 시그널 처리 루틴이 계속 호출되어야 할 필요가 있을 때는 이 루틴들을 스택처럼 쌓아서, 한 핸들러를 빠져나오면 다음 핸들러가 호출되고, 마지막으로 정리용 루틴이 호출되도록 시그널 처리를 최적화한다.

5.2 파이프(Pipe)

일반적으로 사용하는 리눅스 쉘들은 모두 리다이렉션(redirection)을 지원한다. 예를 들어

```
$ ls | pr | lpr
```

이라는 명령은 ls 명령이 출력하는 파일 이름들을 pr 명령의 표준 입력으로 보내고, pr 명령은 입력된 내용을 페이지 단위로 나눈다. pr 명령의 표준 출력으로 나온 결과는 다시 lpr 명령의 표준 입력으로 보내져서 기본 프린터로 출력된다. 파이프는 위의 예에서처럼 한 프로세스의 표준 출력을 다른 프로세스의 표준 입력으로 보내주는 단방향 바이트 스트림이다. 파이프로 연결되는 프로세스들은 이런 리다이렉션이 일어나고 있다는 것은 알지 못하며, 보통 때와 마찬가지로 동작한다. 여기서 프로세스간에 임시 파이프를 만들어 연결시켜주는 것은 쉘이다.

include/linux/inode_fs_i.h 참조

리눅스에서 파이프는 임시로 만들어진 VFS inode를 똑같이 가리키는 두 개의 `file` 자료구조를 사용해서 구현되며, 여기서 VFS inode는 메모리상의 물리적 페이지를 가리키게 된다[35]. 그림 5.1은 각 `file` 자료구조가 각기 다른 파일 연산 루틴 벡터를 가리키는 포인터를 가지고 있는 모습을 보여준다. 여기서 한 `file` 자료구조는 파이프에 쓰는 함수에 대한 포인터를, 다른 자료구조는 파이프에서 읽어들이는 함수에 대한 포인터를 가진다. 이것은 보통의 파일에 읽고 쓰는 시스템 콜이 아래 계층의 차이에 관계없이 동작하도록 한다[36]. 쓰는 프로세스가 파이프에 쓴 데이터는 공유 데이터 페이지에 복사되고, 읽는 프로세스가 그 파이프로부터 읽어 들일때는 공유 데이터 페이지로부터 데이터가 복사되게 된다. 리눅스는 파이프

역주 34) 이러한 문제는 시그널이 발생했는지 확인하는 것이 커널 모드에서이고, 시그널 핸들러는 사용자 모드에서 실행되어야 하기 때문에, 커널 모드에서 바로 시그널 핸들러는 부를 수 없어서 발생한다. 이에 사용자 모드로 바꾸어 시그널 핸들러를 부르고 이것이 끝났을 때 마치 시스템 콜을 한 것처럼 커널 모드로 다시 돌아오게 하는 방법을 쓰는 것이다. (flyduck)

역주 35) 즉 파이프를 위해 파일 시스템에 파일을 생성하는 것이 아니라, 메모리 상에 공유 메모리 페이지를 만들어서 VFS inode가 이를 사용하게 하는 것이며, 이는 파이프의 속도를 빠르게 한다. (flyduck)

역주 36) VFS inode의 `f_op` 항목은 읽기, 쓰기를 포함하여 함수에 관련된 연산들의 포인터 배열이며, 파일 연산은 이 항목에 있는 함수 포인터를 부름으로써 이루어진다. 여기서 파이프를 읽어들이는 파일을 타나내는 `file` 자료구조는 보통 파일에 읽는 함수에 대한 포인터가 아니라 파이프에서 읽어들이는 포인터를 가리키게 하더라도, 시스템 콜에 있어서는 어짜피 `f_op` 항목의 함수 포인터를 이용하므로 아무런 차이가 없는 것이다. 이것은 파이프에 쓰는 `file` 자료구조에 있어서도 마찬가지며, 이러한 기법은 리눅스 커널 곳곳에서 쓰이고 있다. (flyduck)

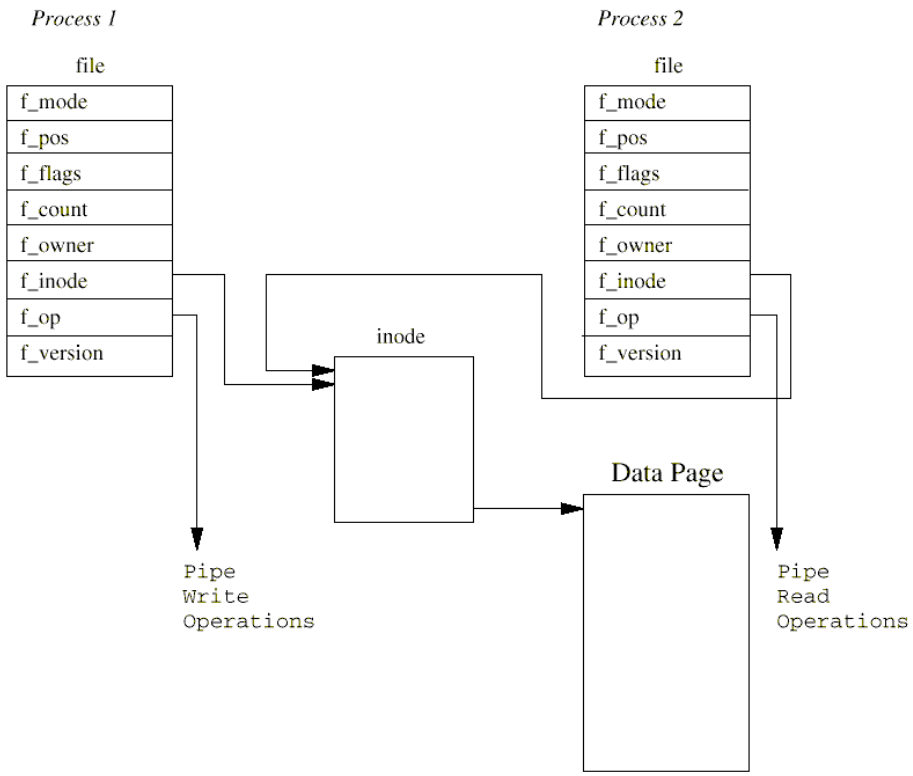


그림 5.1 : 파이프

에 대한 접근을 동기화해야 한다. 파이프의 읽는 프로세스와 쓰는 프로세스가 반드시 차례를 지킬 수 있도록 해야 하고, 그렇게 하기 위해 락(lock)과 대기큐(waiting queue), 시그널 등을 사용한다.

fs/pipe.c pipe_write() 참조

프로세스가 파이프에 쓰기를 할 때는 쓰기를 하는 표준 라이브러리 함수를 사용한다. 이들 함수들에는 파일 기술자(file descriptor)를 넘기는데, 이는 프로세스가 가진 여러개의 `file` 자료구조(이들 각각은 프로세스가 열어 놓은 파일을 나타내며, 이 경우에는 열어 놓은 파이프를 나타낸다)에 대한 인덱스이다[37]. 리눅스 시스템 콜은 이 파이프를 나타내는 `file` 자료구조에서 가리키고 있는 쓰기 루틴을 사용한다. 이 쓰기 루틴은 쓰기 요청을 처리하기 위해 파이프를 나타내는 VFS inode에 있는 정보들을 이용한다. 파이프에 요청한 바이트들을 모두 쓸 공간이 있고, 파이프를 읽는 프로세스가 락을 걸어두지 않았다면, 리눅스는 먼저 파이프에 락을 걸고, 쓸 데이터 바이트들을 프로세스의 주소공간에서 공유 데이터 페이지로 복사한다. 만약 읽는 프로세스가 파이프에 락을 걸어두었거나 데이터를 담을 충분한 공간이 없다면, 현재 프로세스는 해당 파이프 inode에 있는 대기큐에 들어가 잠들고, 실행할 수 있는 다른 프로세스를 선택하기 위해 스케쥴러를 호출한다. 잠든 프로세스는 인터럽트 허용 상태이므로, 시그널을 받을 수 있으며, 읽는 프로세스에 의해 쓸 데이터를 담기에 충분한 공간이 생기거나 파이프의 락을 풀리면 깨어나게 된다. 데이터를 쓰고 나면 파이프의 VFS inode의 락을 풀고, inode의 대기큐에서 기다리며 잠들어 있는 읽는 프로세스를 깨우게 된다.

fs/pipe.c pipe_read() 참조

파이프에서 데이터를 읽는 과정은 파이프에 쓰는 과정과 매우 비슷하다. 프로세스들은 블럭킹을 하지 않고 읽을 수 있는데 (이는 파일이나 파이프를 열 때 어떤 모드를 사용하였느냐에 따라 다르다[38]), 이 경우 읽을 데이터가 없거나 파이프에 락이 걸려있으면 에러가 돌아온

역주 37) `task_struct`는 파일에 관련된 자료구조인 `struct files_struct files` 항목을 가지고 있으며, `files_struct`는 `struct file *fd[NR_OPEN]` 항목을 가지고 있다. 이 `fd`에서의 인덱스가 파일 기술자이며, `fd` 항목은 프로세스에서 열은 파일에 대한 정보를 가지고 있다. (flyduck)

역주 38) `file` 자료구조의 `f_flags` 항목이 파일에 관련된 플래그를 가지고 있는데, 여기에

다. 이는 프로세스가 잠들지 않고 실행을 계속할 수 있다는 것이다. 블럭킹 모드라면 파이프 inode의 대기큐에서 쓰기 프로세스가 끝나기를 기다려야 한다. 양쪽 프로세스가 파이프를 통한 작업을 종료하면, 파이프 inode는 공유 데이터 페이지와 함께 폐기된다.

리눅스는 지정 파이프(named pipe)도 지원한다. 지정 파이프는 FIFO라고도 불리는데 이는 파이프가 먼저 들어온 것이 먼저 나가는(First In First Out, FIFO) 원칙에 따라 동작하기 때문이다. 파이프에 먼저 쓴 데이터는 파이프에서 읽을 때 먼저 나온다. 파이프와 달리 FIFO는 임시적으로 생성된 것이 아니라 파일 시스템에 실재 존재하는 것이며, `mkfifo` 명령으로 생성할 수 있다. 프로세스는 해당하는 접근 권한을 가지고 있다면 FIFO를 자유롭게 사용할 수 있다. FIFO를 여는 방법은 파이프와는 조금 다르다. 파이프(두개의 `file` 자료구조와 이들이 가진 VFS inode, 공유 데이터 페이지)는 한번에 만들어지는데 반해, FIFO는 이미 존재하는 것이며, 사용자에 의해 열고 닫혀지는 것이다[39]. 리눅스는 FIFO에 쓰는 프로세스가 없을 때 다른 프로세스가 이를 읽기 위해 열려고 하는 것이나, FIFO에 쓰는 프로세스가 FIFO에 쓰기를 하기 전에 읽는 프로세스가 읽으려고 하는 것 모두 처리해야 한다. 이를 제외하면, FIFO는 거의 완전히 파이프와 똑같은 방법으로 취급되며, 같은 자료구조와 연산을 사용한다[40].

5.3 소켓(Socket)[41]

REVIEW NOTE : 네트워크 장을 쓴 다음에 추가한다.

5.3.1. 시스템 V IPC 메커니즘

리눅스는 유닉스 System V (1983)에서 처음 등장한 세가지 종류의 프로세스간 통신 방법을 제공한다. 이들은 메시지 큐(message queue)와 세마포어(semaphore), 그리고 공유 메모리(shared memory)이다. 이들 시스템 V IPC 방법들은 모두 똑같은 인증 방법을 공유한다. 프로세스는 커널에 시스템 콜로 이들 자원을 가리키는 유일한 참조 식별자(reference identifier)를 전달함으로써만 이들에 접근할 수 있다. 이들 시스템 V IPC 객체들에 대한 접근은 접근 권한(access permission)을 가지고 검사하는데, 파일에 대한 접근을 검사하는 것과 많이 비슷하다. 시스템 V IPC 객체에 대한 접근 권한은 시스템 콜을 통하여 객체의 생성자에 의해 지정된다. 각 통신 방법들은 참조 식별자를 자원 테이블에 대한 인덱스처럼 사용하는데, 참조 식별자는 말그대로 인덱스인 것은 아니고, 인덱스를 만들기 위해서는 약간의 계산이 필요하다.

시스템에 있는 시스템 V IPC 객체를 나타내는 리눅스 자료구조는 모두, 프로세스의 소유자와 생성자의 uid, gid와 이 객체에 대한 접근 모드(소유자, 그룹, 그밖에 대한)와 IPC 객체의 키를 가진 `ipc_perm`이라는 자료구조를 포함하고 있다. 키는 시스템 V IPC 객체의 참조 식별자를 찾는 한 방법으로 쓰인다. 모두 두 종류의 키를 지원하는데, 공용(public)와 개인용

include/linux/ipc.h 참조

`O_NONBLOCK` 가 지정되어 있으면 블럭킹을 하지 않는 상태이다. 이는 파일을 열때 지정할 수도, ioctl이나 fcntl같은 시스템 콜을 통해서 바꿀 수도 있다. (flyduck)

역주 39) 앞의 `ls | pr`을 지정 파이프를 사용한다면, 우선 `mkfifo fifo` (파일이름은 다르게 지정해도 된다) 명령으로 지정 파이프를 만든 후, `ls > fifo & pr < fifo`로 하면 된다. (flyduck)

역주 40) FIFO도 파이프와 마찬가지로 실제로 파일을 통하여 통신하는 것이 아니라 메모리에 공유 페이지를 만든다. 즉 읽고 쓰는 것은 파이프와 똑같은 방법을 사용하게 된다. FIFO와 파이프의 차이점은 파이프는 임시로 생성되는 것인데 반해 FIFO는 이미 만들어져 있는 것이므로 두 개의 프로세스가 `open()`, `close()` 함수를 통하여 파일처럼 열 수 있으므로 쉘이 관여하지 않아도 통신할 수 있다는 점이다. (flyduck)

역주 41) 소켓은 네트워킹에서 이야기하는 소켓이다. 소켓은 한 컴퓨터 내에서 프로세스 사이에 통신을 할 수 있게 할 뿐만 아니라 네트웍에 있는 다른 컴퓨터에 있는 프로세스와도 통신을 가능하게 한다. 소켓의 사용은 유닉스에서 전통적으로 사용하는 파일 기술자(file descriptor)를 통하여 한다. 즉 `socket()` 함수를 부르면 소켓을 나타내는 파일 기술자가 돌아오고 이를 가지고 `bind, listen, connect, accept` 등의 연산을 할 수 있으며, 파일과 마찬가지로 `read, write, close` 연산을 할 수 있다. (flyduck)

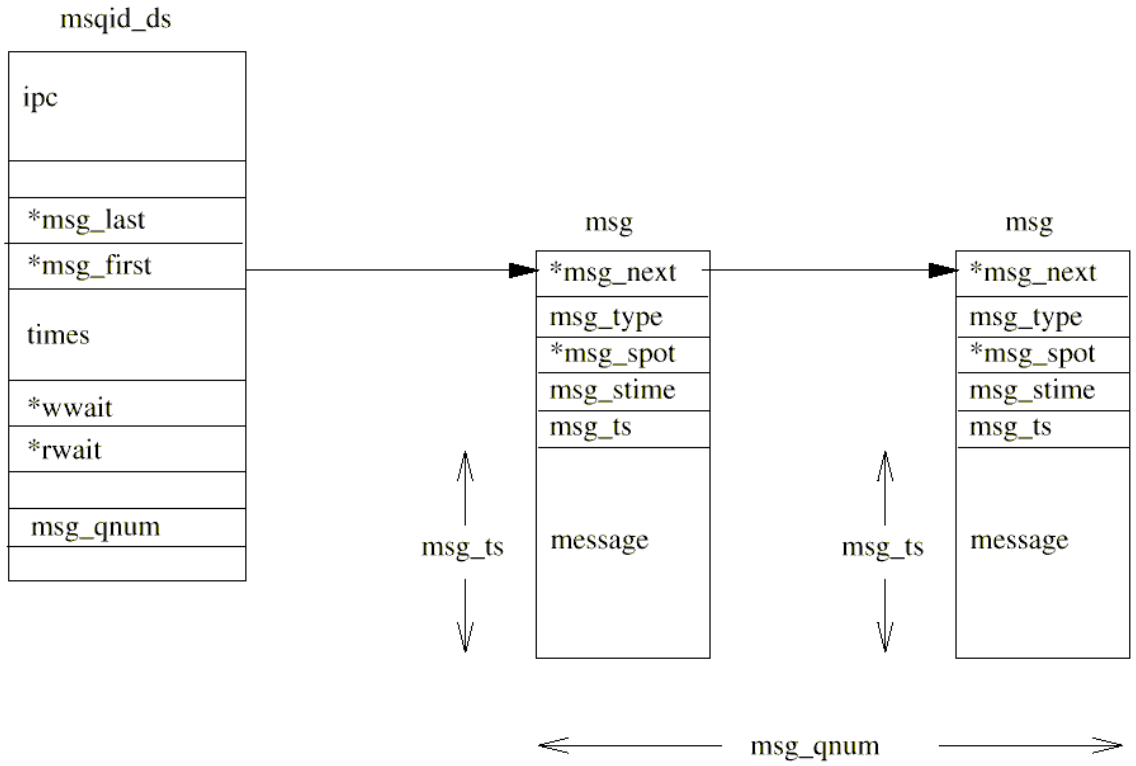


그림 5.2 : 시스템 V IPC 메시지 큐

(private)이 그것이다. 만약 키가 공용라면 시스템에 있는 어떤 프로세스든지 권한 검사를 통과한다면 시스템 V IPC 객체에 대한 참조 식별자를 찾을 수 있다[42]. 시스템 V IPC 객체는 키로 참조할 수 없으며, 이들에 대한 참조 식별자로만 참조할 수 있다.

5.3.2 메시지 큐(Message Queue)

메시지 큐는 하나 이상의 프로세스가 메시지를 쓸 수 있고, 이를 하나 이상의 프로세스가 읽을 수 있을 수 있도록 한다. 리눅스는 메시지 큐의 리스트를 `msgque` 벡터로 관리한다. `msgque`의 각 원소는 메시지 큐에 대한 모든 것을 기술하는 `msqid_ds` 자료구조를 가리킨다. 메시지 큐를 하나 생성하면 `msqid_ds` 자료구조를 시스템 메모리에서 할당받아 이 벡터에 삽입한다.

include/linux/msg.h 참조

각 `msqid_ds` 자료구조는 `ipc_perm` 자료구조와, 이 큐에 들어온 메시지에 대한 포인터들을 가지고 있다. 추가로, 리눅스는 큐에 마지막으로 쓴 시간같은 큐 수정 시간도 유지한다. `msqid_ds`는 두 개의 대기큐도 가지고 있다 : 하나는 큐에 쓰려는 프로세스를 위해, 하나는 큐에서 읽을려는 프로세스를 위해서다.

프로세스가 큐에 메시지를 쓰려고 할 때마다, 효력 사용자 식별자(effective user identifier)와 효력 그룹 식별자(effective group identifier)를 큐의 `ipc_perm` 자료구조에 있는 모드와 비교한다. 그래서 프로세스가 큐에 쓸 수 있다면 메시지는 프로세스의 주소공간에서 `msg` 자료구조로 복사되고 메시지 큐의 마지막에 놓인다. 각 메시지에는 같이 협동하는 프로세스간에 서로 약속한 타입인, 응용프로그램 지정 타입을 꼬리표로 단다. 리눅스는 쓸 수 있는 메시지의 개수와 길이를 제한하고 있으므로 메시지를 쓸 공간이 없을 수도 있다. 이런 경우 프로세스는 메시지 큐의 쓰기 대기큐(`msqid_ds`의 `*wwait` 항목)에 추가되고 실행할 새로운 프

역주 42) 공용키가 아니라 개인용키를 사용하는 경우는 `ipc_perm`의 `key` 항목이 `IPC_PRIVATE`로 지정되는데, 이를 가지고 참조 식별자를 찾을 수 없다. 개인용키를 사용하는 IPC 개체는 개체 번호를 통해서만 접근할 수 있다. (flyduck)

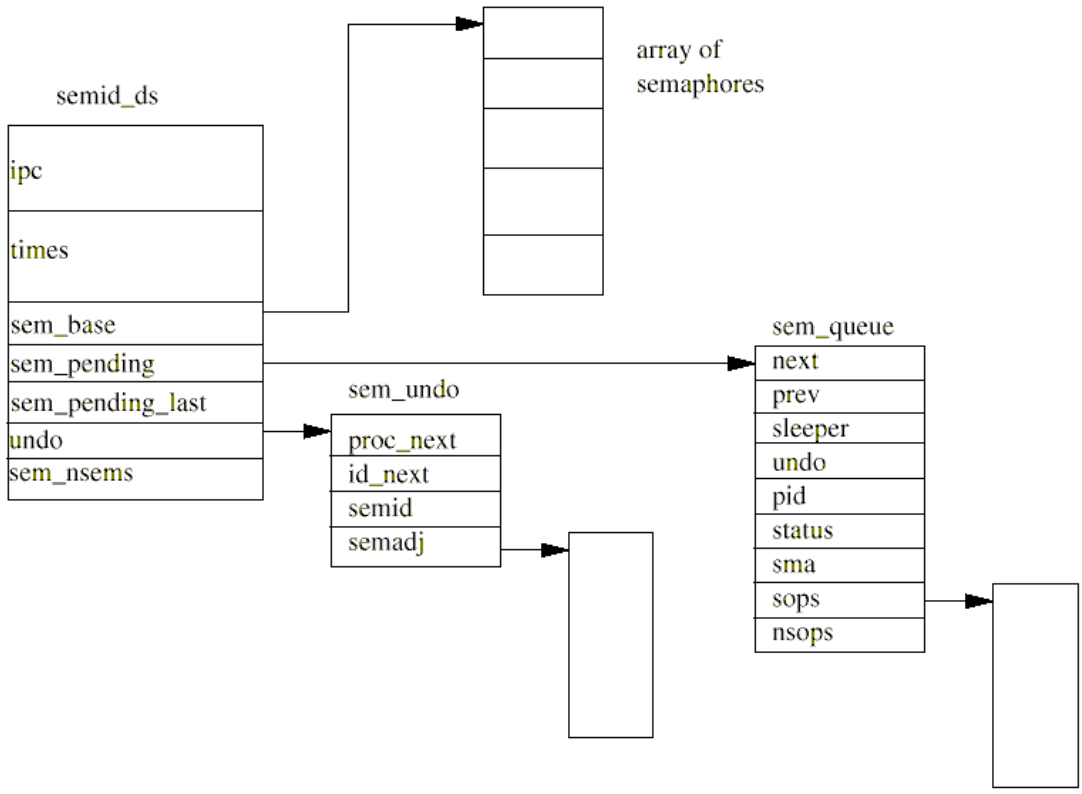


그림 5.3 : 시스템 V IPC 세마포어

로세스를 선택하기 위해 스케쥴러를 호출한다. 프로세스는 메시지 큐에서 하나 이상의 메시지가 읽혔을 때 깨어나게 된다.

큐에서 읽는 것은 비슷한 과정을 거친다. 마찬가지로 프로세스가 가진 큐에 대한 접근 권한을 검사한다. 읽는 프로세스는 타입에 관계없이 큐에 있는 첫번째 메시지를 가져올 지, 또는 특정한 타입을 가진 메시지를 선택할 지 고를 수 있다. 이 기준에 맞는 메시지가 없다면 읽으려는 프로세스는 메시지 큐의 읽기 대기큐(`msgq_id`의 `*rwait` 항목)에 추가되고, 스케쥴러가 실행된다. 큐에 새로운 메시지를 쓰게 되면 이 프로세스는 깨어나 다시 실행할 수 있게 된다.

5.3.3 세마포어(Semaphore)

세마포어의 가장 단순한 형태는 메모리의 한 위치에 있는 변수로, 그 값을 하나 이상의 프로세스가 검사하고 설정(test and set)할 수 있는 것이다. 이 검사 및 설정(test and set) 연산은, 각 프로세스에 있어서, 중단될 수 없는, 즉 원자성을 가진 것이다. 즉 한번 시작되면 아무것도 이를 중단할 수 없다[43]. 이 검사 및 설정 연산의 결과는 세마포어의 현재값에 더하여 값을 설정하는 것이며, 이 값은 양수일 수도 음수일 수도 있다. 검사 및 설정 연산의 결과에 따라서 한 프로세스는 다른 프로세스가 세마포어의 값을 바꿀 때까지 기다리며 잠들어야 할 수도 있다. 세마포어는 동시에 한 프로세스만이 실행해야 하는 중요한 코드가 있는, 임계지

역주 43) 검사를 하고 설정하는 사이에 중단이 되어 다른 것이 실행되었는데 여기서 이 값을 검사하고 설정한다면, 다시 이전으로 돌아와서 설정하려고 할 때는 이미 값이 바뀐 이후가 되어 문제가 발생할 것이다. 따라서 이 검사 및 설정 연산은 중단되어서는 안되며, CPU에서 제공하는 특별한 명령어를 이용하거나 운영체제 코드를 통하여 구현된다. 리눅스에서는 커널모드에서 비선점형이므로 이 연산이 중단되지 않는다고 생각하고 일반 연산으로 처리한다. 커널 코드에서 세마포어 값을 검사하는 함수는 `try_semop()`이며, 세마포어 값을 바꾸는 함수는 `do_semop()`이다. 이들은 `ipc/sem.c`에 있다. (flyduck)

역(critical region)을 구현하는데 사용할 수 있다.

여러개의 협동하는 프로세스가 하나의 데이터 파일에서 레코드를 읽거나 쓴다고 하자. 이 때 파일에 대한 접근이 완전히 조화롭게 이루어지길 바랄 것이다. 여기서 세마포어를 사용할 수 있는데, 먼저 세마포어의 초기값을 1로 하고, 파일 연산을 하는 코드의 주위에 두개의 세마포어 연산을 두어서, 첫번째 것은 세마포어의 값을 검사하고 값을 감소시키고, 다음 것은 값을 검사하고 증가시키게 할 수 있다. 파일에 접근하려는 첫번째 프로세스는 세마포어의 값을 감소시키려고 하고, 이것이 성공하여 세마포어의 값은 0이 된다. 이 프로세스는 이제 계속 진행하여 데이터 파일을 사용하지만, 이를 사용하려고 하는 다른 프로세스는 세마포어의 값을 감소시키려고 했는데 결과가 -1이 되므로 실패한다. 이 프로세스는 첫번째 프로세스가 데이터 파일 작업을 끝마칠 때가지 중단될 것이다. 첫번째 프로세스가 데이터 파일 작업을 마치면 세파포어의 값을 다시 증가시켜 1로 만든다. 이제 기다리는 프로세스는 깨어나서 이번에는 세마포어를 감소시키려는 시도가 성공하게 된다[44].

include/linux/sem.h 참조

시스템 V IPC 세마포어 객체들은 각각 세마포어의 배열을 나타내고, 리눅스는 이를 나타내기 위해 `semid_ds` 자료구조를 사용한다. `semary`는 시스템에 있는 모든 `semid_ds` 자료구조를 가리키고 있는, 포인터의 벡터이다. `semid_ds` 자료구조에는 `sem_nsems` 갯수만큼의 세마포어 배열이 있으며, 각각은 `sem` 자료구조로 기술된다. 이 세마포어 배열은 `sem_base`이 가리키고 있다[45]. 시스템 V IPC 세마포어 객체의 세마포어 배열을 관리할 수 있는 권한을 가진 모든 프로세스들은 이들을 다루는 시스템 콜을 부를 수 있다. 시스템 콜은 한번에 여러개의 연산을 지정할 수 있으며, 각 연산은 세가지 입력 - 세마포어 인덱스, 연산 값, 플래그들의 세트 - 으로 나타내진다[46]. 세마포어 인덱스는 세마포어 배열에서의 인덱스이며, 연산 값은 세마포어의 현재 값에 추가될 숫자 값이다. 먼저 리눅스는 모든 연산이 성공할 수 있는지 테스트한다. 연산 값을 세마포어의 현재 값에 더한 값이 0 이상이거나, 연산 값과 세마포어의 현재 값이 모두 0일 때, 이 연산은 성공하게 된다. 만약 세마포어 연산의 하나라도 실패한다면 리눅스는 프로세스를 중단할 수 있는데, 이는 시스템 콜을 부를 때 플래그에 블럭킹 모드를 사용하지 않을거라고 지정하지 않은 경우이다. 프로세스가 중단되어야 한다면 리눅스는 수행해야 할 세마포어 연산의 상태를 저장하고, 현재 프로세스를 대기큐에 넣는다. 이 작업은 `sem_queue` 자료구조를 스택에 만들어 이것의 내용을 채움으로써 이루어진다[47]. 새 `sem_queue` 자료구조는 세마포어 객체의 대기 큐의 끝에 놓여진다 (여기서 `sem_pending`과 `sem_pending_last` 포인터를 사용한다). 현재 프로세스는 `sem_queue` 자료구조에 있는 대기큐(`sleeper` 항목)에 놓여지고, 실행할 다른 프로세스를 고르기 위해 스케쥴러가 호출된다.

만약 모든 세마포어 연산이 성공하여 프로세스가 중단될 필요가 없다면, 리눅스는 계속 진행하여 세마포어 배열의 올바른 멤버에게 연산을 적용한다. 리눅스는 이제 대기큐에서 기다리며 중단되어 있는 프로세스들이 이 세마포어 연산에 적용될 수 있는지 검사해야 한다. 리

역주 44) 첫번째 프로세스가 세마포어의 값을 0으로 바꾼 후에는 다음 프로세스가 검사단계에서 실패하므로 세마포어의 값은 변하지 않으며, 첫번째 프로세스가 이 값을 1로 바꾼 후에야 기다리고 있던 프로세스가 이를 바꿀 수 있게 된다. 그래서 세마포어의 값은 항상 0보다 크거나 같게 된다. (flyduck)

역주 45) 하나의 세마포어 객체에 여러개의 세마포어 배열(sem의 배열)이 있으며, 시스템 콜은 이 중의 일부만을 검사하고 설정할 수 있다. 임계지역에서 여러개의 세마포어를 필요로하는 경우 이 중에 필요로 하는 것들만 지정할 수 있다. (flyduck)

역주 46) 세마포어 연산을 하는 시스템 콜은 `sys_semop(int semid, struct sembuf *sops, unsigned nsops)`이며, 여기서 하나의 연산을 가리키는 `sembuf`는 `sem_num, sem_op, sem_flg` 세가지 원소로 이루어져 있다. 이 세마포어 연산은 세마포어 값을 감소시킬 수도, 증가시킬 수도 있다. (flyduck)

역주 47) 메시지 큐의 경우는 대기큐가 `task_struct`를 가지고 있는 단순한 `wait_queue`의 연결 리스트로 되어 있지만, 세마포어의 대기큐는 `sem_queue`의 연결 리스트로 되어 있다. 이는 메시지 큐의 경우 읽기를 기다리는지, 쓰기를 기다리는지만 구별하면 되지만, 세마포어에서는 세마포어 연산으로 넘겨준 인자들을 모두 저장하고 있어야 하고 좀 더 복잡한 연산이 필요하기 때문에 이 정보를 모두 `sem_queue`에 저장하는 것이다.(flyduck)

눅스는 연산 미결큐(sem_pending)의 각 멤버를 차례로 살펴보고, 이번엔 세마포어 연산이 성공할 수 있는지 알아보기 위한 테스트를 한다. 만약 성공한다면 연산 미결 리스트에서 sem_queue 자료구조를 제거하고 세마포어 배열에 그 세마포어 연산을 적용한다. 리눅스는 잠든 프로세스를 깨워 다음번 스케쥴러가 실행될 때에는 다시 시작할 수 있도록 만든다. 리눅스는 미결 리스트를 처음부터 시작하여 더이상 세마포어 연산을 적용할 수 없고, 깨울 프로세스가 없을 때까지 계속 살펴본다.

세마포어에는 한가지 문제가 있는데 데드락(deadlock)이 바로 그것이다. 이는 한 프로세스가 임계지역에 들어가면서 세마포어의 값을 바꾸었는데 프로세스가 잘못되거나 강제로 종료되어서 이 임계지역을 빠져나가지 못한 경우에 발생한다[48]. 리눅스는 이런 문제를 세마포어 배열에 대한 조정 리스트를 관리함으로써 막는다. 이 개념은 이런 조정을 적용하면 세마포어가 그 프로세스가 세마포어 연산을 수행하기 이전의 상태로 되돌아가게 하는 것이다. 조정에 대한 것은 sem_undo 자료구조에 보관되고, 이들은 semid_ds 자료구조와 세마포어 배열을 사용하는 프로세스의 task_struct 양쪽에 큐된다.

각 개별적인 세마포어 연산은 조정을 관리하도록 요구할 수 있다. 리눅스는 프로세스마다 각 세마포어 배열에 대해 많아봐야 하나의 sem_undo 자료구조를 관리한다. 만약 연산을 요청한 프로세스가 이 자료구조를 가지고 있지 않다면 필요할 때 하나 생성할 것이다. 새로 만들어진 sem_undo 자료구조는 이 프로세스의 task_struct 자료구조와 세마포어 배열의 semid_ds 자료구조 양쪽에 큐된다. 세마포어 배열에 있는 세마포어에 연산을 적용하면 연산값을 반대로 한 값이 이 프로세스의 sem_undo 자료구조에 있는 조정 배열의 세마포어 엔트리로 추가된다. 즉 연산값이 2를 더하는 것이었다면 이 세마포어의 조정 엔트리에는 -2가 더해진다.

프로세스가 종료하여 지워질 때, 리눅스는 sem_undo 자료구조 세트를 가지고 세마포어 배열에 조정을 적용한다. 만약 한 세마포어 세트가 지워지면 프로세스의 task_struct의 큐되어 있는 sem_undo 자료구조는 그대로 남아있지만, 세마포어 배열 식별자는 잘못된 것일 것이다. 이 경우 세마포어 정리 코드는 간단하게 sem_undo 자료구조를 무시한다.

5.3.4. 공유 메모리(Shared Memory)

공유 메모리는 하나 이상의 프로세스들이 자신들의 가상 주소 공간에 공통으로 나타나는 메모리를 통하여 통신할 수 있도록 한다. 이들 프로세스의 페이지 테이블 각각에는 이 공유 가상 메모리 페이지들을 가리키는 페이지 테이블 엔트리가 있게 된다. 이들은 모든 프로세스의 가상 메모리에서 똑같은 주소에 있을 필요는 없다. 다른 시스템 V IPC 객체와 마찬가지로 공유 메모리 영역로의 접근은 키에 의해 제어되고 접근 권한을 검사하게 된다. 하지만 한번 메모리가 공유되고 나면 프로세스들이 이를 어떻게 사용하는지에 대해서 아무런 검사도 하지 않는다. 프로세스들은 다른 방법, 예를 들어 시스템 V 세마포어같은 것을 사용하여 메모리로의 접근을 동기화하여야 한다.

include/linux/shm.h 참조

새로 만들어진 공유 메모리 영역은 shmid_ds 자료구조로 나타낸다. 이들은 shm_segs 벡터에 저장된다. shmid_ds 자료구조는 공유 메모리 영역이 얼마나 큰지, 얼마나 많은 프로세스가 사용하고 있으며, 공유 메모리가 프로세스의 주소공간에 어떻게 매핑되어 있는지에 대한 정보를 가진다. 공유 메모리를 만든 프로세스가 이 메모리에 대한 접근권한과 키가 공

역주 48) 데드락은 이 경우뿐만 아니라 한 프로세스가 필요로 하는 자원을 다른 프로세스가 사용하고 있어 대기 상태로 갔는데, 나중에 그 프로세스가 앞의 프로세스가 점유하고 있는 자원을 필요로 하게 되어 프로세스들이 서로 상대가 자원 사용을 종료하기만을 기다리게 되는 상태도 포함한다. 이것은 단순히 두 프로세스가 아니라 여러 프로세스가 꼬리에 꼬리를 물고 있을 때 복잡하게 이루어질 수 있다. 단순히 세마포어에 국한하여 이야기한다면, 프로세스가 세마포어를 이용하여 임계지역으로 들어간 후 다시 세마포어를 얻으려고 하지 않는다면 이런 문제는 발생하지 않을 것이다. (flyduck)

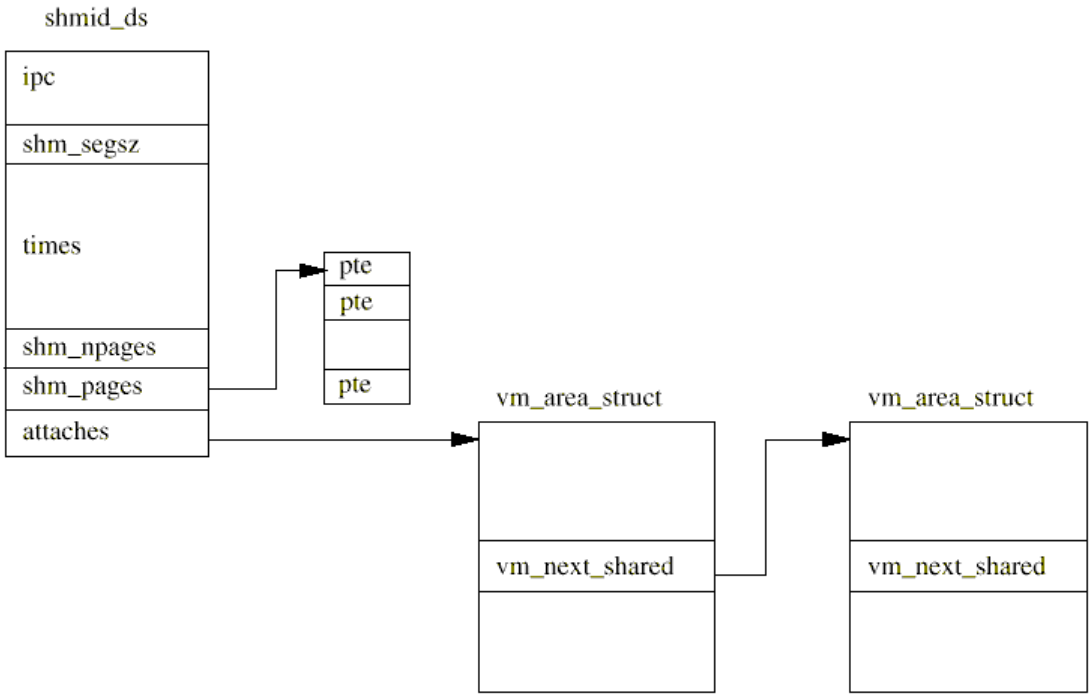


그림 5.4 : 시스템 V IPC 공유 메모리

용인지 개인용인지 제어하며, 충분한 권한만 있다면 공유 메모리를 물리적인 메모리로 락[49] 시킬수도 있다.

메모리를 공유하길 바라는 각 프로세스들은 시스템 콜을 통하여 이 가상 메모리에 연결해야 한다. 이것은 이 프로세스에서의 공유 메모리를 기술하는 새로운 `vm_area_struct` 자료구조를 만들어낸다[50]. 프로세스는 공유 메모리가 자신의 가상 주소 공간에 위치할 곳을 선택할 수도 있고, 아니면 리눅스가 충분히 큰 빈 영역을 선택하도록 할 수도 있다. 새로 만들어진 `vm_area_struct` 자료구조는 `shmid_ds`가 가리키고 있는 `vm_area_struct` 리스트에 추가된다. `vm_newxt_shared`와 `vm_prev_shared` 포인터들은 이들을 서로 연결하는데 사용한다. 가상 메모리는 이렇게 연결하는 동안에 실제로 만들어지지 않으며, 처음으로 프로세스가 여기에 접근하려고 할 때 만들어진다.

프로세스가 공유하고 있는 가상 메모리의 한 페이지에 처음으로 접근을 시도하면 페이지 폴트가 발생한다. 리눅스가 이 페이지 폴트를 처리할 때 이를 기술하는 `vm_area_struct` 자료구조를 발견하게 된다. 여기에는 이 타입의 공유 가상 메모리에 대한 처리 루틴에 대한 포인터가 있다[51]. 공유 메모리의 페이지 폴트 처리 코드는 `shmid_ds`의 페이지 테이블 엔트리를 뒤져서, 공유 가상 메모리의 해당 페이지에 대한 페이지 테이블 엔트리가 있는지 찾는다. 만약 없다면 물리적 메모리를 하나 할당 받아 이를 나타내는 페이지 테이블 엔트리를 만들 것이다. 이를 현재 프로세스의 페이지 테이블에 넣으면서 `shmid_ds`에도 저장한다. 그래서 다음 프로세스가 이 메모리에 접근하려고 하다가 페이지 폴트가 발생하면, 공유 메모리 페이지 폴트 처리 코드가 이를 찾아서, 새로 만들어진 물리적인 페이지를 그 프로세스에게도 사용하게 한다. 따라서 공유 메모리의 어떤 페이지에 접근하는 첫번째 프로세스는 이를 생성하고, 다른 프로세스들이 여기에 접근할 때는 이를 자신의 가상 메모리 공간에 추

역주 49) 가상 메모리를 물리적인 메모리에 존재하게 만드는 것을 말한다. (flyduck)

역주 50) 앞의 프로세스 장에서 설명한 것과 같이 한 프로세스가 할당받은 메모리들은 `vm_area_struct`의 리스트와 AVL 트리로 관리되는데, 이는 페이지 폴트가 발생했을 때 해당 페이지가 실제 프로세스가 사용하는 메모리인지, 어떻게 물리적인 페이지를 만들것인지 알기 위해 사용된다. (flyduck)

역주 51) nopage 연산 (flyduck)

가하게 된다.

프로세스가 더이상 가상 메모리를 공유하길 바라지 않을 때는 여기로의 연결을 끊는다. 이 메모리를 사용하는 다른 프로세스가 존재하는 한은 연결을 끊는 것은 단지 해당 프로세스에게만 영향을 미친다. 그 메모리의 `vm_area_struct`는 `shmid_ds` 자료구조에서 제거되고 해제될 것이며, 프로세스가 공유하는데 사용했던 가상 메모리 영역을 무효한 것으로 나타내기 위해 프로세스의 페이지 테이블이 갱신된다. 마지막으로 메모리를 공유하고 있던 프로세스가 연결을 끊으면 물리적인 메모리에 존재하고 있는 모든 공유 메모리 페이지들은 해제되고, 이 공유 메모리를 나타내던 `shmid_ds` 자료구조도 해제된다.

공유 가상 메모리가 물리적인 메모리로 락되어 있지 않을 때 약간 복잡한 문제가 발생한다. 이는 메모리의 사용량이 많아서 공유 메모리가 스왑 디스크로 스왑된 것인 경우도 있다. 공유 메모리가 어떻게 물리적인 메모리에서 스왑되어 나가거나 들어오는지는 3장에서 설명하고 있다.

번역 : 이승, 이호, 김진석, 김기용, 심마로
정리 : 이호

6장

PCI

PCI(Peripheral Component Interconnect, 주변장치 상호연결)는 이름 그대로, 시스템에 있는 여러 주변장치들을 어떻게 구조적이고 관리를 잘 할 수 있는 방식으로 함께 연결할 것인지를 정의하고 있는 표준이다. 이 표준[3, PCI 로컬버스 규약]은 시스템 장치들을 전기적으로 연결하는 방식과, 각 장치들이 동작해야 하는 방식을 규정하고 있다. 이 장에서는 리눅스 커널이 시스템의 PCI 버스들과 장치들을 초기화하는 방법을 간단히 살펴보도록 한다[52].

그림 6.1은 일반적인 PCI 기반 시스템의 논리 구성도이다. PCI 버스와 PCI-PCI 브릿지는 시스템 장치들을 서로 연결하는 접착제와 같은 것이다. CPU는 첫번째 PCI 버스인 0번 PCI 버스에 연결되어 있고, 이 구성도에서는 비디오 장치가 여기에 연결되어 있다. PCI-PCI 브릿지는 특별한 PCI 장치로서, 1차(primary) PCI 버스와 2차(secondary) PCI 버스인 1번 PCI 버스를 연결한다[53]. PCI 규약에 나오는 전문용어로는, 1번 PCI 버스를 PCI-PCI 브릿지의 다운스트림(downstream), 0번 버스를 브릿지의 업스트림(upstream)이라고 한다[54]. 이 구성도에서 2차 PCI 버스에는 SCSI 카드나 이더넷 카드 등이 연결된다. 물리적으로 브릿지와 2차 PCI 버스, 여기 연결된 두 장치들은 하나의 복합 PCI 카드로 만들 수 있다[55]. PCI-ISA 브릿지는 오래전부터 사용되어온 ISA 장치들을 지원하는 것으로, 이 구성도에서는 키보드와 마우스, 플로피 드라이브를 제어하는 슈퍼 I/O 콘트롤러 칩이 여기에 연결되어 있다.

6.1 PCI 주소공간(PCI Address Space)

역주 52) 이 장의 내용과 분량은 일반적으로 운영체제에 대해 가지는 관심에 비해 자세하고 많은 편이다. 이는 PCI BIOS가 모든 일을 처리하는 인텔 기반 PC와는 달리, 다른 시스템에서는 PCI 버스를 운영체제가 직접 제어를 해야하며, 저자가 주로 알파 AXP 기반 시스템에서 작업을 했기 때문인 것 같다. 하지만 버스 구조에 대한 이해는 실제 하드웨어와 관련된 작업에 있어서 큰 도움을 주리라 생각한다. (flyduck)

역주 53) 하나의 PCI 버스가 감당할 수 있는 장치의 갯수가 한정되어 있기 때문에, 더 많은 장치를 연결하려면 버스를 추가하고 이를 PCI-PCI 브릿지로 연결해야 한다. 예전에는 일반 PC에는 PCI-PCI 브릿지가 없고, 서버용 기계에만 PCI-PCI 브릿지가 있었지만, 요즘에는 일반 PC 용으로도 PCI-PCI 브릿지가 있는 보드가 나오고 있다. MS Windows 운영체제를 사용하고 있다면 시스템 등록정보의 장치관리자에서 시스템 장치에 어떤 것이 연결되어 있는지 확인해보면 좋을 것이다. CPU 버스와 PCI 버스를 연결하는 브릿지와, PCI-ISA 브릿지, PCI-PCI 브릿지 등을 확인할 수 있을 것이다. (flyduck)

역주 54) CPU의 입장에서 생각한다면 0번 버스가 CPU에 바로 연결된 것이기 때문에, 1번 버스에서 0번 버스로 가는 것은 위로 가는 것이므로 업스트림, 0번 버스에서 1번 버스로 가는 것은 다운스트림이 된다. (flyduck)

역주 55) PCI 규약에 따라 하나의 PCI 카드가 최대 8개의 기능을 가질수 있다. (flyduck)

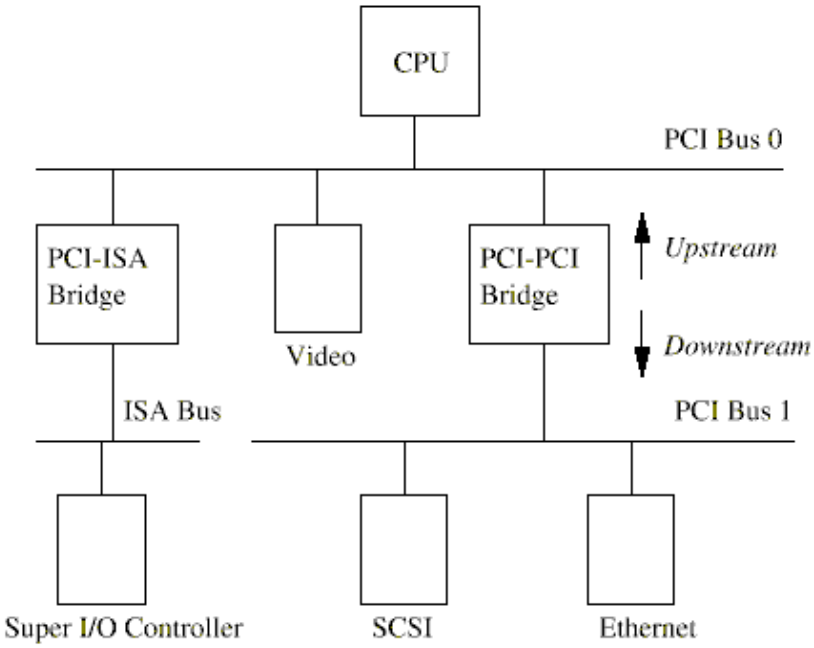


그림 6.1 : PCI 기반 시스템의 예

CPU와 PCI 장치들은 그들이 공유하고 있는 메모리에 접근할 필요가 있다. 이 메모리는 디바이스 드라이버가 PCI 카드를 제어하고 서로 정보를 교환하는데에 사용되는 것이다. 일반적으로 이 공유 메모리에는 장치에 속한 제어 레지스터(control register)와 상태 레지스터(status register)가 들어 있다. 이 레지스터들은 장치를 제어하고 상태를 읽는데 쓰인다. 예를 들어, PCI SCSI 디바이스 드라이버는 SCSI 디스크로 데이터를 쓰려고 할 때, 장치의 상태 레지스터를 읽어서 장치가 쓸 준비가 되었는지 알아 내고, 이 값이 켜져 있을 때에 장치가 원하는 동작을 하도록 제어 레지스터에 값을 쓰게 된다.

CPU가 관리하는 시스템 메모리를 이런 공유 메모리로 사용될 수도 있겠지만, 이렇게 한다면 PCI 장치가 메모리에 접근할 때마다 CPU가 메모리에 접근하지 못한 채 PCI 장치가 작업을 끝마치기를 기다려야 하는 문제가 발생할 것이다. 이는 일반적으로 메모리에 접근하는 것은 동시에 하나로 제한되어 있기 때문이며, 이렇게 한다면 시스템이 느려질 것이다. 그렇다고 주변장치가 메인 메모리에 아무런 통제 없이 접근할 수 있도록 하는 것 역시 좋지 않은 생각이다. 이것은 매우 위험하며, 잘못 만들어진 장치는 시스템을 매우 불안하게 만들 수 있다.

주변장치들은 각자 자신만의 메모리 공간을 가지고 있다. CPU는 이 영역에 자유롭게 접근할 수 있지만, 반대로 이 장치가 시스템 메모리에 접근하는 것은 DMA(Direct Memory Access, 직접 메모리 접근) 채널을 이용하는 경우로만 엄격히 제한되어 있다. ISA 장치는 ISA I/O와 ISA 메모리라는 두가지 주소공간을 가진다. PCI는 세가지 주소공간을 가지는데, PCI I/O, PCI 메모리, 그리고 PCI 설정공간(configuration space)이 그것이다. CPU는 이들 주소공간 모두에 접근할 수 있는데, 디바이스 드라이버는 PCI I/O와 PCI 메모리 주소공간을 사용하며, PCI 설정공간은 리눅스 커널의 PCI 초기화 코드에서 사용하고 있다[56].

역주 56) 처음 IBM PC가 나올 때부터 여기에는 메모리 공간외에 I/O 공간에 있는 포트라는 것이 있었다. 이는 보통의 메모리 접근 명령이 아닌 특수한 포트 입출력 명령을 사용했는데 인텔 CPU에 있는 in, out 명령이 그것이다. ISA 카드에서는 장치의 레지스터에 접근하는데 이런 포트 I/O를 사용하였고, 큰 영역의 데이터에 접근할 때는(예를 들어 그래픽 카드의 메모리) 메모리 영역을 사용하였다. 하지만 인텔 CPU와는 달리 많은 CPU들은 포트 I/O를 지원하지 않는다. 비록 PCI 규약이 I/O 공간을 지원하기 하지만, 이를 사용하지 않는 PCI 카드들도 있으며, 여기서는 칩의 레지스터들이 모두 메모리 공간에 존재한다. ISA 카드에서는 64KB의 I/O 영역과 640KB-1MB, 15MB-16MB(이 영역을 사용하는 장치는

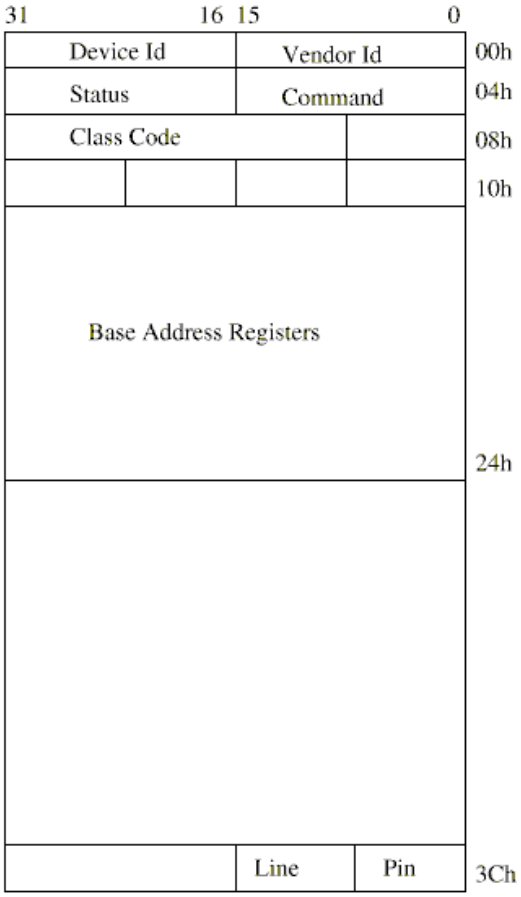


그림 6.2 : PCI 설정 헤더

알파 AXP 프로세서는 원래 시스템 주소공간을 제외한 다른 주소공간에 접근할 수 없도록 되어 있다. 그래서 여기서는 PCI 설정 주소공간과 같은 다른 주소공간에 접근할 수 있도록 해주는 여러 칩셋을 사용한다. 그리고 거대한 가상 주소공간에서 일부를 빼내 PCI 주소공간으로 맵핑하는 희소 주소 매핑(sparse address mapping)방법을 사용한다.

6.2 PCI 설정 헤더(PCI Configuration Header)

모든 PCI 장치는 (PCI-PCI 브릿지도 포함하여) PCI 설정 주소공간 어딘가에 설정에 관련된 자료구조를 가지고 있다. PCI 설정 헤더는 시스템이 장치를 구별하고 제어할 수 있게 한다. 이 헤더가 있는 정확한 위치는 PCI 배치도에서 장치가 위치한 곳에 따라 결정된다. 예를 들어, PCI 비디오 카드를 PC 메인보드에 있는 여러 PCI 슬롯 중 하나에 꽂을 때, 어떤 슬롯에 꽂느냐에 따라 PCI 설정 주소공간에서 각기 다른 위치에 헤더가 위치하게 된다. 이것은 그다지 문제가 되지 않는데, 왜냐면 PCI 장치와 브릿지가 어디 있든간에, 시스템은 설정 헤더에 있는 상태 레지스터, 설정 레지스터를 이용해 그들을 찾아내 설정할 것이기 때문이다.

보통 PCI 설정 헤더의 오프셋는 보드에서의 슬롯 번호에 관련이 있다. 그래서, 첫번째 슬롯의 PCI 설정 헤더가 0번 오프셋에 위치 한다면, 두번째 슬롯의 헤더는 256번 오프셋에 위치하고 (모든 헤더는 똑같이 256바이트 크기이다), 다른 슬롯의 헤더도 이런 식으로 위치하게

아주 드물다)의 메모리 영역을 사용하고 있다. 이는 지금까지 유효한데, 몇가지 문제를 야기하고 있다. PCI에서는 4GB의 I/O 공간과, 32비트 또는 64비트의 메모리 공간을 제공한다. (flyduck)

된다. 시스템별로 PCI 설정 영역에 접근하는 하드웨어 메커니즘이 다르게 정의되어 있으며, 이를 이용하여 PCI 설정을 하는 코드는 주어진 PCI 버스에 있어서 가능한 모든 PCI 설정 헤더를 검사하여, 어떤 장치가 있고 어떤 장치가 없는지를 헤더의 한 항목을 (보통 제작자 식별자(Vendor Identification)[57]항목) 읽고 에러를 검출함으로써 파악한다. [3, PCI 로컬 버스 규약]에서는 빈 PCI slot의 제작자 식별자나 장치 식별자(Device Identification)를 읽으려고 하면 *0xFFFFFFFF*을 돌려주는 것으로 에러 검출방법을 정의하고 있다.

include/linux/pci.h 참조

그림 6.2는 256 바이트의 PCI 설정 헤더의 배치도를 보여준다. 여기에는 다음과 같은 항목이 있다.

제작자 식별자(Vendor Identification) PCI 장치의 제작자를 나타내는 고유번호. 예를 들어 디지털(Digital)은 *0x1011*, 인텔은 *0x8086*를 고유번호로 갖는다.

장치 식별자(Device Identification) 장치 자체를 나타내는 고유번호. 예를 들어 디지털의 21141 고속 이더넷 장치는 *0x0009* 값을 갖는다.

상태(Status) 이 항목은 장치의 상태를 나타내는데, 각각의 비트들이 갖는 의미는 표준에서 정의하고 있다. [3, PCI 로컬 버스 규약]

.

명령(Command) 시스템은 이 항목에 값을 씀으로써, 장치의 PCI I/O 메모리를 접근을 허가하는 것 같은, 장치를 제어하는 일을 한다.

분류코드(Class Code) 이 장치가 속한 장치의 유형을 구별한다. 모든 종류의 장치에 대해 비디오, SCSI 같은 식의 표준 분류가 있다. SCSI에 대한 분류코드는 *0x0100*이다.

베이스 주소 레지스터(Base Address Register) 이 레지스터는 장치가 사용하는 PCI I/O, PCI 메모리 공간의 유형과 크기, 위치를 지정하는데 사용된다.

인터럽트 핀(Interrupt Pin) PCI 카드에 있는 물리적인 핀 중 네 개는, 카드로부터 PCI 버스로 인터럽트를 전달하는 역할을 한다. 표준에서는 이들을 각각 A, B, C, D라고 부른다[58]. 인터럽트 핀 항목은 PCI 장치가 이들 핀 중 어떤 핀을 사용하고 있는지 나타낸다. 보통 특정 장치에 있어서 인터럽트 핀은 직접 배선되어 있다. 즉, 시스템이 부팅할 때마다 그 장치는 똑같은 인터럽트 핀을 사용한다는 것이다. 이 정보는 인터럽트 처리 시스템이 장치에서 발생하는 인터럽트를 관리할 수 있도록 해준다.

인터럽트 라인(Interrupt Line) 이 항목은 PCI 초기화 코드와 디바이스 드라이버, 리눅스의 인터럽트 처리 서브시스템 사이에 인터럽트 핸들을 전달하기 위해 사용한다. 여기 있는 값은 디바이스 드라이버에겐 의미가 없겠지만, 인터럽트 핸들러가 PCI 장치로부터 온 인터럽트를 리눅스 운영체제에 있는 올바른 디바이스 드라이버의 인터럽트 처리 코드로 인터럽트를 전달할 수 있게 한다. 리눅스가 인터럽트를 처리하는 방법에 대해 자세한 것은 7장을 참조하라.

6.3 PCI I/O와 PCI 메모리 주소

이들 두 주소공간은 장치가 CPU 상의 리눅스 커널에서 실행되는 디바이스 드라이버와 통신하기 위해 사용하는 곳이다. 예를 들어, DECchip 21141 고속 이더넷 장치는 자신의 내부 레지스터를 PCI I/O 공간에 매핑한다. 그러면 해당하는 리눅스 디바이스 드라이버는 장치를 제어하기 위해서 이들 레지스터를 읽고 쓴다. 비디오 드라이버는 비디오 정보를 저장하기 위해 방대한 양의 PCI 메모리 공간을 사용한다.

역주 57) 여기서 제작자는, PCI 카드의 제작자라기 보다는 카드에 있는 PCI와 연결 역할을 하는 칩의 제작자이다. (flyduck)

역주 58) 이 이름은 IRQ A-D와 무관하며, PCI 카드상의 핀에 대한 이름이다. (flyduck)

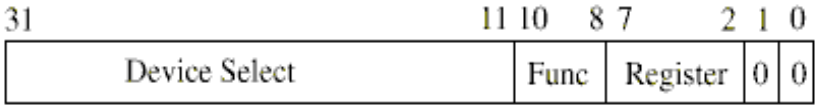


그림 6.3 : 0 번 타입 설정 사이클

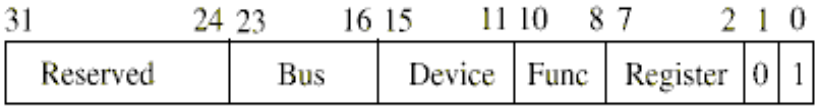


그림 6.4 : 1 번 타입 설정 사이클

이들은 PCI 시스템이 셋업이 되고, PCI 설정 헤더에 있는 명령(Command) 항목에서 이들 주소공간에 대한 접근을 허용할 때까지, 아무도 이 공간에 접근할 수 없다. 여기서 PCI를 설정하는 코드만이 PCI 설정 영역을 읽고 쓸 수 있으며, 리눅스 디바이스 드라이버는 단지 PCI I/O와 PCI 메모리 공간만 읽고 쓸 수 있다는 사실을 기억하기 바란다.

6.4 PCI-ISA 브릿지(Bridge)

이 브릿지는 PCI I/O, PCI 메모리 공간으로의 접근을 ISA I/O, ISA 메모리 공간으로의 접근으로 바꾸어 줌으로써, 오래전부터 사용해온 ISA 장치를 지원하는 역할을 한다. 지금 팔리는 많은 시스템들은 PCI 버스 슬롯과 함께 여러개의 ISA 버스 슬롯을 가지고 있는데, 시간이 지날수록 이전 것과 호환성을 유지할 필요는 줄어들고, 언젠가는 PCI 슬롯만 있는 시스템이 팔릴 것이다[59]. ISA 주소공간(I/O와 메모리 공간)에서 ISA 장치의 레지스터가 위치해 있는 곳은, 안개가 자욱한 시절에 나온 초창기 8080 기반 PC에 의해 고정되었다. 5000 달러가 넘는 알파 AXP 기반 컴퓨터조차도 ISA 플로피 콘트롤러는 처음 나온 IBM PC에서와 똑같은 I/O 공간을 사용한다[60]. PCI 규약에서는 이 문제를 PCI I/O와 메모리 주소공간에서 아래쪽 영역을 ISA 시스템의 ISA 주변장치 용으로 예약을 하고, 하나의 PCI-ISA 브릿지를 통해 PCI 메모리로의 접근을 이 영역으로 바꾸어 줌으로써 해결한다.

6.5 PCI-PCI 브릿지

PCI-PCI 브릿지는 시스템에 있는 PCI 버스들을 붙여주는 특별한 PCI 장치이다. 간단한 시스템에는 PCI 버스가 하나밖에 없지만, 하나의 PCI 버스가 지원할 수 있는 PCI 장치의 개수에는 전기적인 제한이 있어서, 더 많은 PCI 장치를 지원하기 위해서 PCI-PCI 브릿지를 통해 PCI 버스를 추가할 수 있도록 한다. 이는 특히 고성능 서버에 있어서 중요하다. 당연히, 리눅스는 PCI-PCI 브릿지를 지원한다.

6.5.1 PCI-PCI 브릿지 : PCI I/O와 PCI 메모리 윈도우(Memory Window)

PCI-PCI 브릿지는 다운스트림으로 가는 PCI I/O나 PCI 메모리에 읽거나 쓰는 요청 중의 일

역주 59) PCI가 등장한 초창기에는 빠른 입출력 속도를 필요로 하는 장치들만 PCI 카드로 나왔지만, PCI의 여러 장점으로 인해 지금은 거의 모든 카드가 PCI 용으로 제작되고 있다. ISA 카드는 제작자의 입장에서 만들기 쉽다는 장점이 있지만, 사용자의 입장에서는 설정하기가 까다롭다는 단점 때문에 갈수록 찾아보기 힘들어지고 있다. (flyduck)

역주 60) ISA 장치의 가장 큰 단점은 사용하는 I/O 공간과 메모리 공간, IRQ 등이 하드웨어적으로 고정되어 있다는 점이다. 몇 장치들은 관습처럼 고정되어 그대로 이어져오고 있으며, 다른 장치들은 하드웨어에 있는 점퍼로 설정하거나, EPROM에 설정값을 기록해놓고 있다. (flyduck)

부만들 통과시킨다. 예를 들어 그림 6.1에서, 0번 PCI 버스에서 1번 PCI 버스로 가는 읽고 쓰는 명령이 있을 때, PCI-PCI 브릿지는 그 주소가 SCSI 카드나 이더넷 카드의 메모리 일 때에만 통과시켜주고, 그 외의 주소일 때는 무시해버린다. 이런 필터링은 필요없는 주소가 시스템 전체로 전달되는 것을 막아준다. 이를 위해 PCI-PCI 브릿지가 1차 버스(primary bus)에서 2차 버스(secondary bus)로 전달해야 하는 PCI I/O와 PCI 메모리 공간의 베이스 주소와 범위를 프로그램해야 한다. 한번 PCI-PCI 브릿지를 설정하고 나면, 디바이스 드라이버가 이 윈도우를 통해서 PCI I/O와 PCI 메모리 공간에 접근하는 한, PCI-PCI 브릿지는 보이지 않는다. 이는 PCI 디바이스 드라이버 제작자들을 편하게 해주는 중요한 특징이다. 나중에 살펴볼 것이지만, 이는 리눅스가 PCI-PCI 브릿지를 설정하는 것을 좀 까다롭게 한다.

6.5.2 PCI-PCI 브릿지 : PCI 설정 사이클(Configuration Cycle)과 PCI 버스에 번호 붙이기

PCI 초기화 코드가 메인 PCI 버스(0번 PCI 버스)에 있지 않는 장치들에 접근할 수 있으려면, 브릿지가 자신의 1차 인터페이스에서 2차 인터페이스로 설정 사이클[61]를 넘길건지 말건지를 결정하도록 할 수 있는 메커니즘이 있어야 한다. 사이클이란 PCI 버스에서 보이는 주소이다. PCI 규약에서는 두가지 형식의 PCI 설정 주소를 정의하고 있다. 이는 0번 타입과 1번 타입인데 그림 6.3과 6.4에서 비교해서 보여주고 있다. 0번 타입 PCI 설정 사이클에는 버스 번호가 없고, 모든 장치는 이를 같은 PCI 버스에 대한 PCI 설정 주소로 해석한다. 0번 타입 설정 사이클에서 11-31번 비트는 장치 선택 항목이다. 시스템을 디자인하는 방법중 하나는 각각 다른 장치에 다른 비트를 부여하는 것인데, 여기서는 비트 11은 0번 슬롯에 있는 PCI 장치를, 비트 12는 1번 슬롯에 있는 PCI 장치를 선택한다. 다른 방법은 장치의 슬롯 번호를 바로 11-31 비트에 써 넣는 것이다. 이 중 어떤 방법을 사용하는지는 시스템의 PCI 메모리 컨트롤러에 따라 다르다.

1번 타입 PCI 설정 사이클에는 PCI 버스 번호가 있으며, 이 타입의 설정 사이클은 PCI-PCI 브릿지만 사용하고 다른 장치들은 모두 무시한다. 모든 PCI-PCI 브릿지는 1번 타입 설정 사이클을 보았을 때, 그것을 자신의 PCI 버스의 다운스트림으로 통과시킬지를 결정할 수 있다. PCI-PCI 브릿지가 1번 타입 설정 사이클을 무시할건지, 아니면 다운스트림 PCI 버스로 통과시킬 것인지는, PCI-PCI가 어떻게 설정되었느냐에 달려있다. 모든 PCI-PCI 브릿지는 1차 버스 인터페이스 번호와 2차 버스 인터페이스 번호를 가지고 있다. 1차 버스 인터페이스는 CPU에 더 가까운 쪽에 있는 버스이고, 2차 버스 인터페이스는 더 멀리 있는 것이다. 또한 각 PCI-PCI 브릿지는 종속 버스(subordinate bus) 번호를 가지고 있는데, 이는 2차 버스 인터페이스 너머 브릿지로 연결된 모든 PCI 버스에서 최대 버스 번호이다. 다르게 표현하면, 종속 버스 번호는 PCI-PCI 브릿지의 다운스트림으로 연결된 PCI 버스 번호 중 가장 큰 값이다. PCI-PCI 브릿지가 1번 타입 설정 사이클을 만나면, 브릿지는 다음 중에서 한가지 일을 한다.

- 지정된 버스 번호가 브릿지의 2차 버스 번호와 종속 버스 번호 사이에 있지 않으면 (이 두 버스 번호도 포함하여) 무시한다.
- 버스 번호가 브릿지의 2차 버스 번호와 일치하면 0번 타입 설정 명령으로 변환한다.
- 지정된 버스 번호가 2차 버스 번호보다 크고 종속 버스 번호보다 작거나 같으면, 바꾸지 않고 그대로 2차 버스 인터페이스로 통과시킨다.

따라서 그림 6.9에 있는 배치도에서 3번 버스에 있는 장치1을 지정하고자 한다면, CPU로부터 1번 타입 설정 명령을 만들어야 한다. 그러면 브릿지1은 이를 바꾸지 않고 1번 버스로 통과시키고, 브릿지2는 이를 무시하며, 브릿지3은 이를 0번 타입 설정 명령으로 바꾸고 3번 버스로 보내 장치1이 응답하게 된다.

PCI 설정을 할 때 버스에 번호를 할당하는 것은 개별 운영체제에 따라 다르겠지만, 어떤 방식으로 번호를 붙이든간에 다음 명제는 시스템에 있는 모든 PCI-PCI 브릿지에 있어서 참이

역주 61) 설정 사이클이란 PCI가 초기화가 되지 않았을 때 PCI 장치들을 설정하기 위하여 사용하는 특별한 주소를 말한다. (flyduck)

어야 한다.

"PCI-PCI 브릿지 너머에 있는 모든 PCI 버스의 번호는, 2차 버스 번호와 종속 버스 번호 (이들을 포함하여) 사이에 있어야 한다."

만약 이 규칙이 깨진다면, PCI-PCI 브릿지는 1번 타입 설정 사이클을 통과시키지 않거나 제대로 변환하지 못할 것이고, 시스템은 자신에 있는 PCI 장치를 찾지 못하거나 초기화하지 못할 것이다. 리눅스는 번호를 올바로 붙이기 위해서 이들 특별한 장치들(PCI-PCI 브릿지)을 특정한 순서로 설정한다. 리눅스에서 PCI 브릿지와 버스에 번호를 붙이는 방식은 6.6.2장에서 실제 예제와 함께 설명한다.

6.6 리눅스 PCI 초기화

리눅스에서 PCI 초기화 코드는 다음 세가지 논리적인 부분으로 쪼갤 수 있다.

PCI 디바이스 드라이버 이 유사 디바이스 드라이버(pseudo device driver)[62]는 0번 버스부터 PCI 시스템을 찾기 시작하여, 시스템에 있는 모든 PCI 장치와 브릿지들을 찾는다. 그리고 해당 자료구조의 연결 리스트를 만들어 시스템의 배치도를 나타낸다. 더불어, 찾은 모든 브릿지에 번호를 부여한다.

drivers/pci/pci.c include/linux/ pci.h 참조

PCI BIOS 이 소프트웨어 계층은 [4, PCI BIOS ROM 규약]에 기술된 서비스들을 제공한다. 알파 AXP에서는 이런 BIOS 서비스가 없지만, 똑같은 일을 하는 동등한 코드가 리눅스 커널에 포함되어 있다.

arch/*/kernel/ bios32.c 참조

PCI 확정(PCI Fixup) 시스템별로 다른 이 코드는 시스템별로 다른 PCI 초기화의 잡다한 종료부분을 깨끗하게 마무리한다.

arch/*/kernel/ bios32.c 참조

6.6.1 리눅스 커널 PCI 자료구조

리눅스 커널은 PCI 시스템을 초기화하면서 시스템의 실제 PCI 배치도를 그대로 나타내는 자료구조를 만든다. 그림 6.5는 그림 6.1에서 예로 든 PCI 시스템에 대해 만들어지는 자료구조를 보여준다.

각 PCI 장치는 (PCI-PCI 브릿지도 포함하여) `pci_dev` 자료구조로, PCI 버스는 `pci_bus` 자료구조로 나타낸다. 최종결과는 버스들의 트리 구조로 트리의 각 노드는 자신에 연결된 여러 PCI 장치들을 자식으로 가진다. PCI버스는 PCI-PCI 브릿지를 통해서만 도달할 수 있으므로 (첫번째 PCI 버스인 0번 버스를 제외하고), 각 `pci_bus` 자료구조는 접근할 때 거쳐야 하는 PCI-PCI 브릿지에 대한 포인터를 갖는다. 이 PCI 장치는 PCI 버스의 부모 PCI 버스의 자식이다.

그림 6.5에는 나오지 않지만 시스템에 있는 모든 PCI 장치에 대한 포인터인 `pci_devices`가 있다. 시스템에 있는 모든 PCI 장치는 자신의 `pci_dev` 자료구조를 가지고 있고, 이들은 이 큐(`pci_devices`)에 들어있다. 이 큐는 리눅스 커널이 시스템에 있는 모든 PCI 장치를 빨리 찾는데 사용한다.

6.6.2 PCI 디바이스 드라이버

역주 62) 실제로 디바이스 드라이버인 것이 아니라, 디바이스 드라이버처럼 장치를 구동하는 역할은 하지만 디바이스 드라이버 형태를 갖추지 않은 것이므로 유사 디바이스 드라이버라고 한다. 이런 것으로는 가상 파일 시스템이나 네트웍 드라이버가 있다. (flyduck)

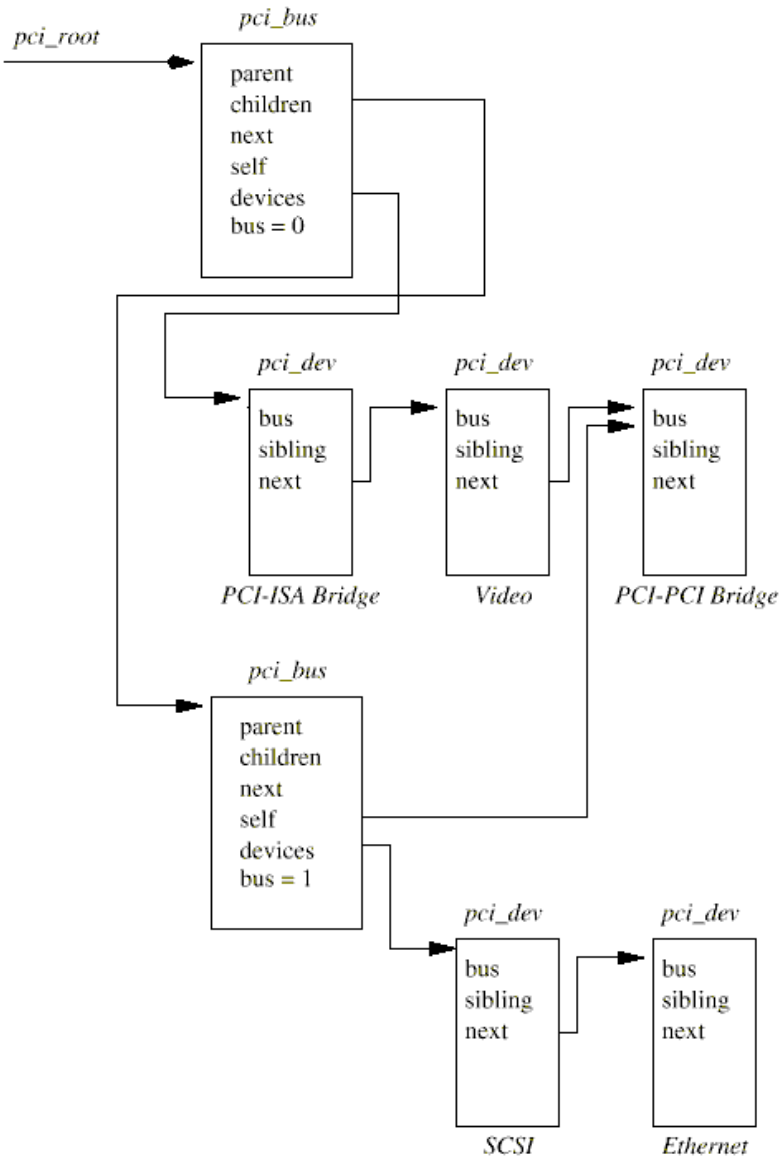


그림 6.5 : 리눅스 커널 PCI 자료구조

drivers/pci/pci.c
Scan_bus() 참조

PCI 디바이스 드라이버는 실제 디바이스 드라이버가 아니라, 시스템 초기화 때 불리는 운영체제의 한 함수이다. PCI 초기화 코드는 시스템에 있는 모든 PCI 버스에서 모든 PCI 장치들(PCI-PCI 브릿지를 포함하여)을 조사해야 한다. 이 코드는 PCI BIOS 코드를 이용하여, 현재 조사하고 있는 PCI 버스의 모든 가능한 슬롯이 점유되어 있는지 아닌지 확인한다. 그리고 그 PCI 슬롯이 점유되어 있으면, 그 장치를 기술하는 `pci_dev` 자료구조를 만들고, 존재하는 PCI 장치의 리스트(`pci_devices`에서 가리키고 있다)에 이를 추가한다

PCI 초기화 코드는 0번 PCI 버스부터 찾기 시작한다. 이 코드는 가능한 모든 PCI 슬롯에서 가능한 모든 PCI 장치의 제작자 식별자(Vendor Identification)와 장치 식별자(Device Identification)를 읽으려고 한다. 그리고 점유되어 있는 슬롯을 발견하면, 그 장치를 나타내는 `pci_dev` 자료구조를 만든다. PCI 초기화 코드에 의해 만들어진 모든 `pci_dev` 자료 구조는 PCI-PCI 브릿지를 포함하여 모두 `pci_devices`라는 단일 연결 리스트에 연결된다.

만약 찾은 PCI 장치가 PCI-PCI 브릿지라면, `pci_bus` 자료구조를 만들어 `pci_bus` 트리와 `pci_root`가 가리키고 있는 `pci_dev` 자료구조에 연결한다. PCI 초기화 코드는 장치의 분류 코드가 *0x060400* 라는 것으로 그 PCI 장치가 PCI-PCI 브릿지임을 알 수 있다. 그리고 나서 리눅스 커널은 자신이 찾은 PCI-PCI 브릿지의 다른 쪽(다운스트림)의 PCI 버스를 설정한

다. 또 다른 PCI-PCI 브릿지를 발견하더라도 똑같은 방법으로 설정한다. 이 과정은 깊이탐색(depthwise) 알고리즘이라고 하는 방법이다. 시스템의 PCI 배치도는 넓이탐색(breadthwise)을 하기 전에 깊이탐색을 통하여 완전히 구성이 된다. 그림 6.1을 보면 리눅스가 0번 PCI 버스에 있는 비디오 장치를 설정하기 전에, 1번 PCI 버스에 있는 이더넷과 SCSI 장치를 설정했음을 알 수 있다.

리눅스가 다운스트림 PCI 버스를 찾을 때, 중간에 있는 PCI 브릿지의 2차 버스 번호와 종속 버스 번호를 설정해야 하는데, 이는 다음에 자세하게 설명하고 있다.

PCI-PCI 브릿지 설정하기 - PCI 버스 번호 부여하기

PCI-PCI 브릿지가, PCI I/O, PCI 메모리, 또는 PCI 설정 주소공간에 읽고 쓰려는 시도를 통과시킬 수 있으려면, 브릿지는 다음과 같은 사항을 알아야 한다 :

1차 버스(primary bus) 번호 PCI-PCI 브릿지에 바로 연결된 업스트림 버스 번호

2차 버스(secondary bus) 번호 PCI-PCI 브릿지에 바로 연결된 다운스트림 버스 번호

종속 버스(subordinate bus) 번호 브릿지의 다운스트림으로 도달할 수 있는 버스들의 가장 큰 번호

PCI I/O와 PCI 메모리 윈도우 PCI-PCI 브릿지의 모든 다운스트림에서 사용하는 PCI I/O 주소공간과 PCI 메모리 주소공간의 윈도우에 대한 베이스 주소와 크기

문제는 어떤 주어진 PCI-PCI 브릿지를 설정하려고 할 때 그 브릿지의 종속 버스 번호를 알 수 없다는 것이다. 다운스트림으로 PCI-PCI 브릿지가 더 있는지 모르고, 안다고 하더라도 그것이 어떤 번호를 갖게 될지도 모른다. 해답은 깊이탐색 재귀 알고리즘을 사용하여, 각 버스에 있는 PCI-PCI 브릿지를 조사하고, 찾으면 번호를 부여하는 것이다. PCI-PCI 브릿지를 찾으면, 2차 버스에 번호를 붙이고, 임시적으로 종속 버스 번호에 *0xFF*를 지정한 후, 다운스트림으로 PCI-PCI 브릿지를 찾아 계속 번호를 붙여나간다. 이것은 복잡해 보이겠지만, 아래에 있는 실제 동작하는 예제를 보면 이 과정이 더 명쾌해질 것이다.

PCI-PCI 브릿지 번호붙이기 : 1 단계 그림 6.6의 배치도에서 처음 찾게 되는 브릿지는 *브릿지1*이다. *브릿지1*의 다운스트림 PCI 버스는 1번이 되며, *브릿지1*의 2차 버스 번호로 1번이, 그리고 임시적으로 종속 버스 번호로 *0xFF*가 할당된다. 이는 PCI 버스 번호로 1번이나 이 이상을 지정한 1번 타입 설정 사이클은 모두 *브릿지1*을 통과하여 PCI 버스 1번으로 가게된다는 것이다. 만약 1번 타입 설정 사이클의 버스 번호가 1번이라면 이는 0번 타입 설정 사이클로 변환이 되겠지만, 다른 버스 번호라면 변환되지 않고 그대로 있을 것이다. 이 과정은 리눅스의 PCI 초기화 코드가 1번 PCI 버스로 진행하여 조사하기 위해 해야 하는 것이다.

PCI-PCI 브릿지 번호붙이기 : 2단계 리눅스는 깊이탐색 알고리즘을 사용하므로, 초기화 코드는 1번 PCI 버스로 진행하여 이를 조사한다. 여기서 PCI-PCI *브릿지2*를 발견하게 되고, PCI-PCI *브릿지2*를 넘어서는 더이상 PCI-PCI 브릿지가 없으므로 종속 버스 번호와 2차 인터페이스 번호로 똑같이 2를 갖게 된다. 그림 6.7은 이 시점에서 버스와 PCI-PCI 브릿지가 어떤 값을 갖게 되는지 보여준다.

PCI-PCI 브릿지 번호붙이기 : 3단계 PCI 초기화 코드는 1번 PCI 버스를 조사하는 곳으로 되돌아와 다른 PCI-PCI 브릿지인 *브릿지3*을 찾게 된다. 이는 1차 버스 번호로 1을, 2차 버스 번호로 3을, 그리고 종속 버스 번호로 *0xFF*를 갖게 된다. 그림 6.8에는 이 때 시스템이 어떻게 설정되는지 보여준다. 이제 버스 번호로 1, 또는 2나 3이 지정된 1번 타입 PCI 설정 사이클은 해당하는 PCI 버스로 올바르게 배달될 것이다.

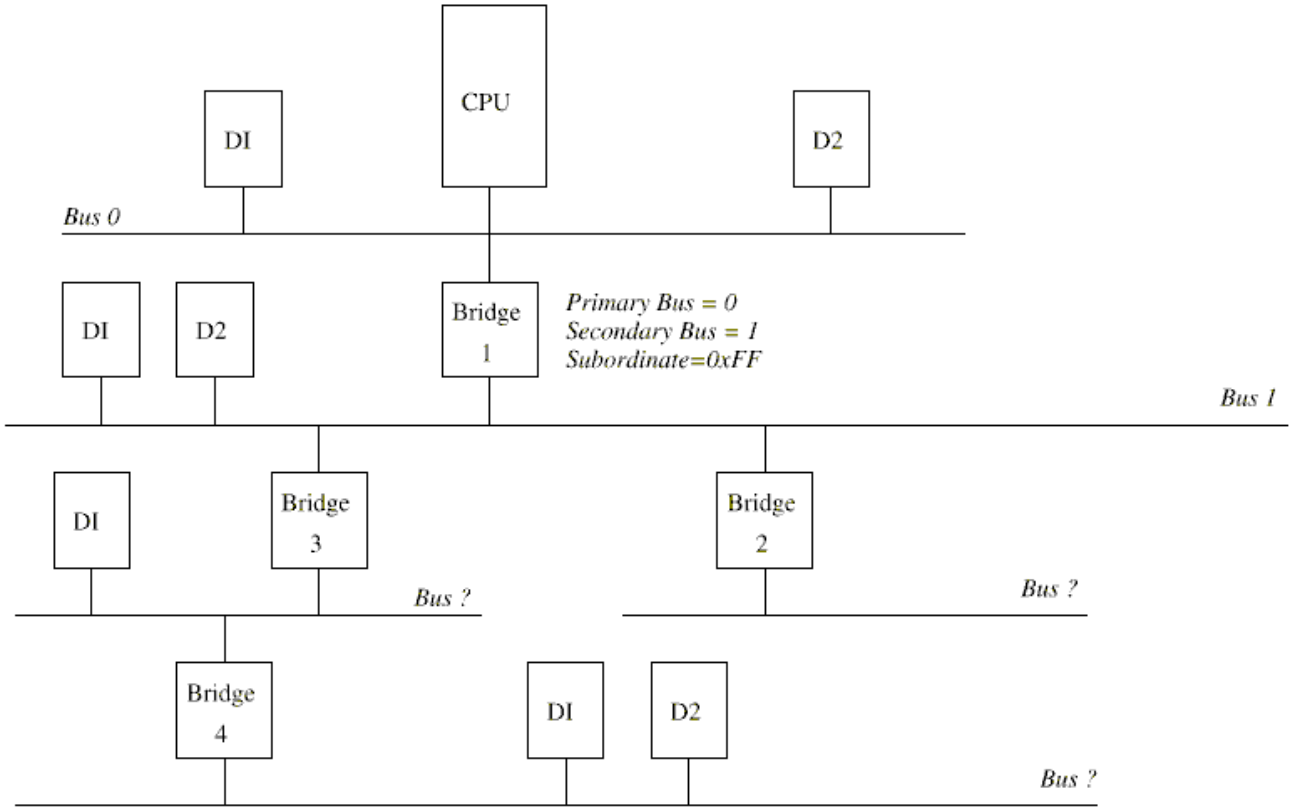


그림 6.6 : PCI 시스템 설정 : 1 단계

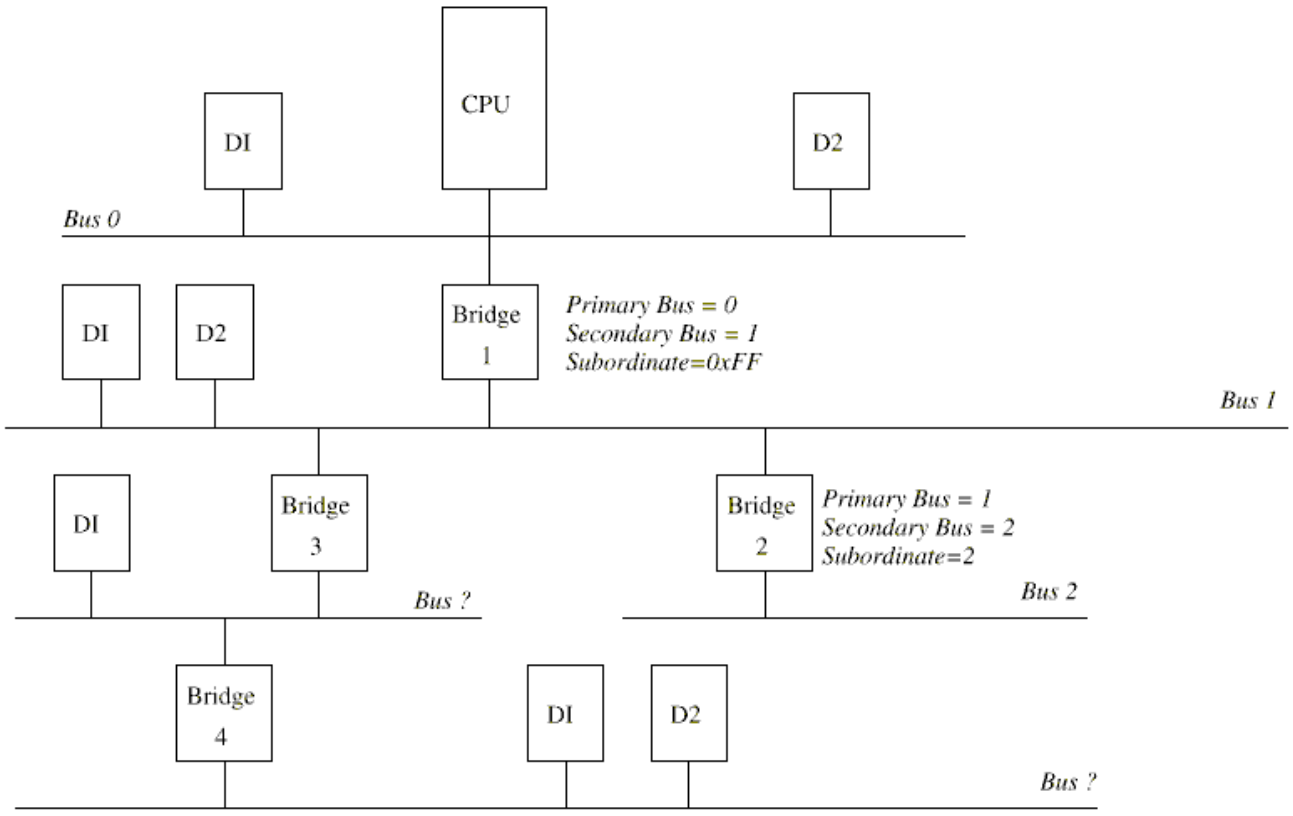


그림 6.7 : PCI 시스템 설정 : 2 단계

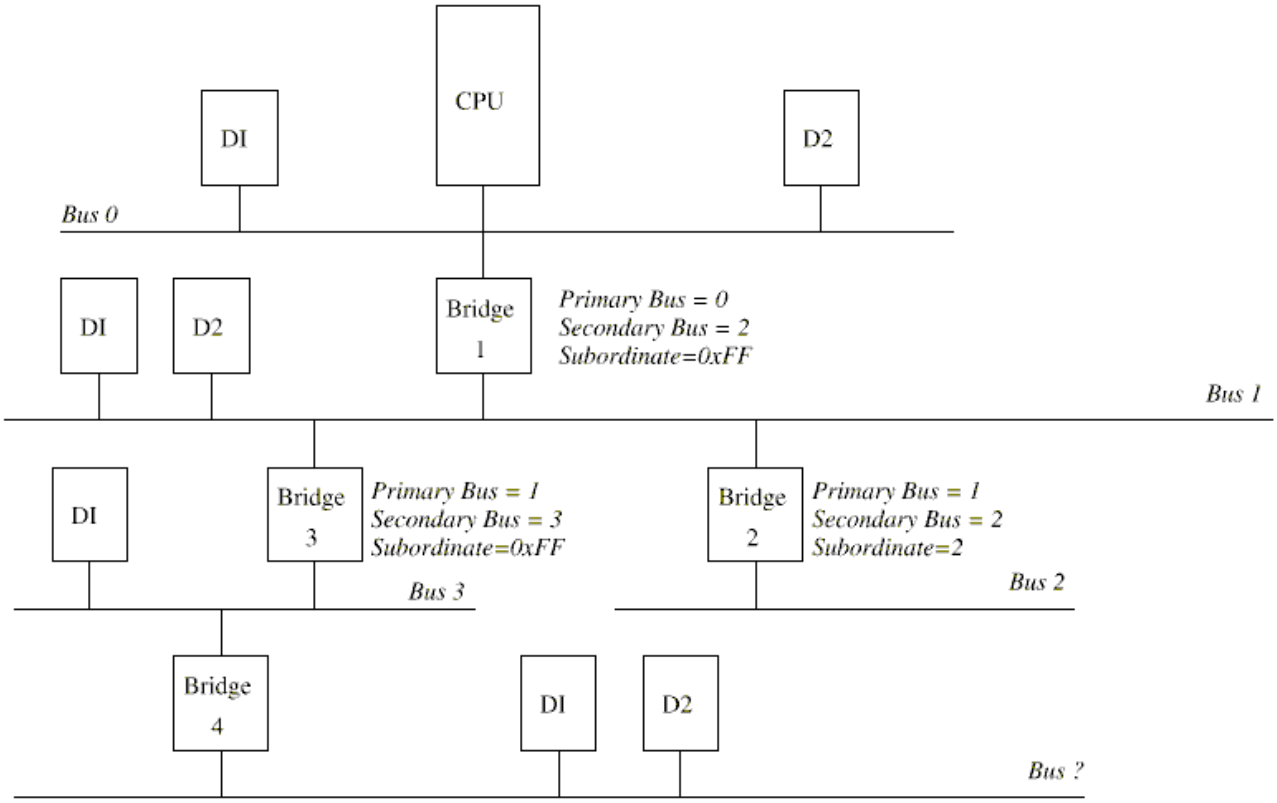


그림 6.8 : PCI 시스템 설정 : 3 단계

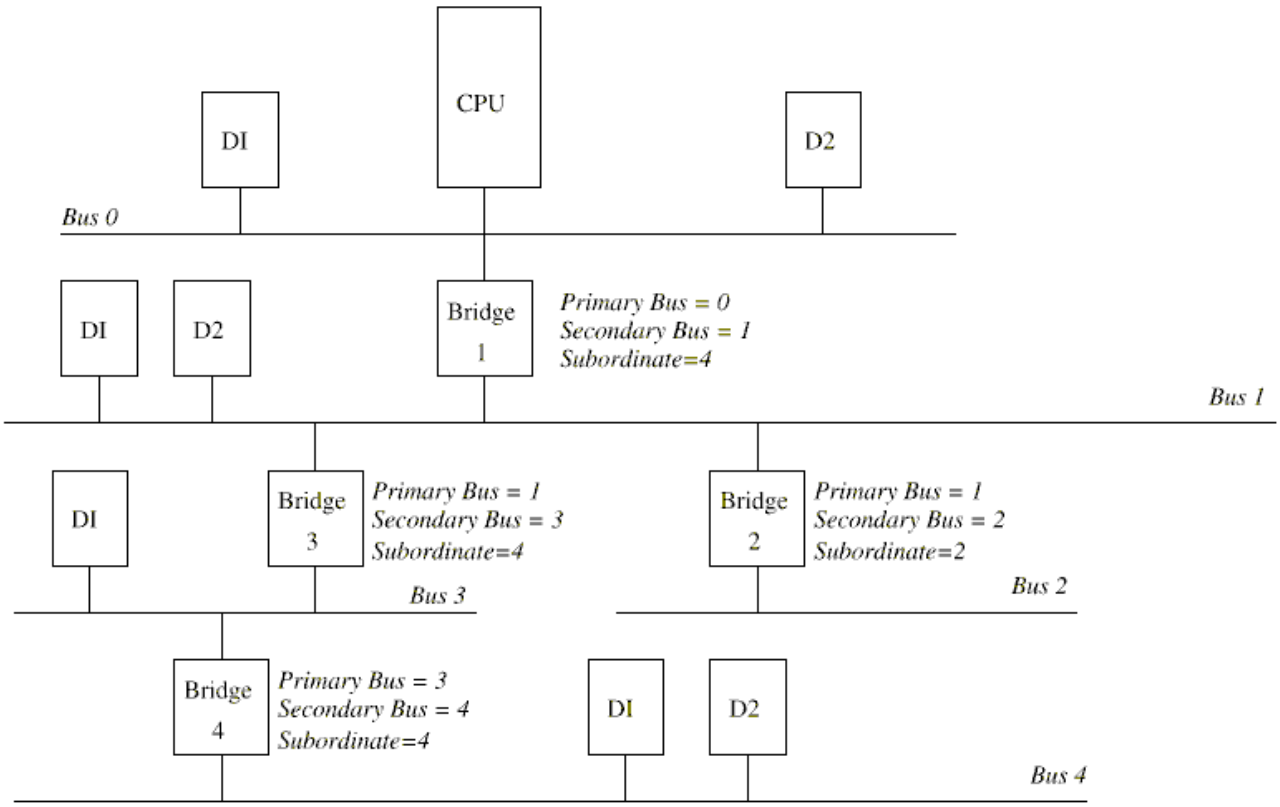


그림 6.9 : PCI 시스템 설정 : 4 단계

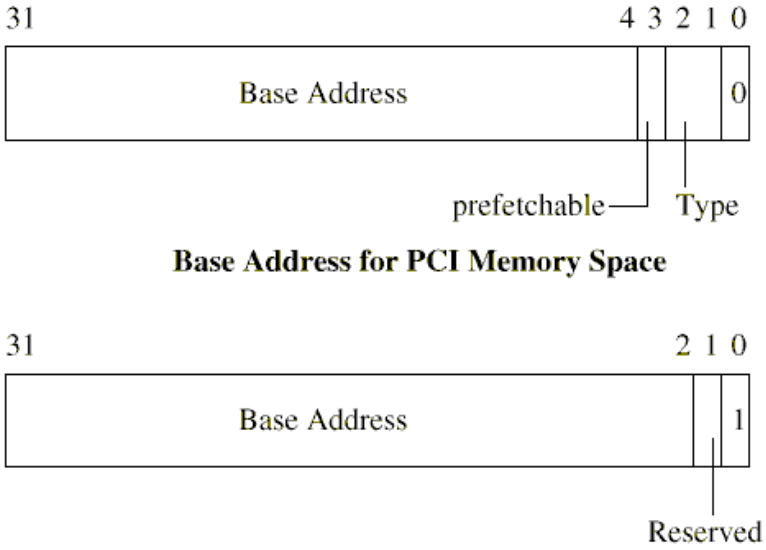


그림 6.10 : PCI 설정 헤더 : 베이스 주소 레지스터들

PCI-PCI 브릿지 번호붙이기 : 4단계 리눅스는 *브릿지3*의 다운스트림인 3번 버스를 조사하기 시작한다. 3번 PCI 버스는 다른 PCI-PCI 브릿지(*브릿지4*)를 갖고 있고, 이는 1차 버스 번호로 3을, 2차 버스 번호로 4를 부여받는다. 이것은 이 줄기에서 가장 끝에 있는 브릿지이므로, 종속 버스 번호로 4를 부여받는다. 초기화 코드는 PCI-PCI *브릿지3*으로 돌아와 종속 버스 번호로 4를 지정한다. 마지막으로 PCI 초기화 코드는 PCI-PCI *브릿지1*의 종속 버스 번호로 4를 할당할 수 있게 된다. 그림 6.9는 최종적인 버스 번호를 보여준다.

6.6.3 PCI BIOS 함수

arch/*/kernel/ bios32.c 참조

PCI BIOS 함수들은 모든 플랫폼들에서 공통적인 표준 함수들 시리즈 중의 하나이다. 예를 들어, 이들은 인텔 기반 시스템과 알파 AXP 기반 시스템에 있어 동일하다. 이들은 CPU가 모든 PCI 주소공간에 제어를 위해 접근할 수 있게 한다. 리눅스 커널 코드와 디바이스 드라이버만이 이를 사용할 수 있다.

6.6.4 PCI 확정(PCI Fixup)

arch/*/kernel/ bios32.c 참조

알파 AXP용 PCI 확정 코드는 인텔용(기본적으로 아무것도 하지 않는 코드이다)보다 훨씬 많다. 인텔 기반 시스템은 부팅시에 실행되는 시스템 BIOS를 가지고 있으며, 이것이 PCI 시스템의 설정을 이미 끝내버렸기 때문이다. 그래서 리눅스는 이미 설정되어 있는 것을 매핑하는 것 외에는 할 일이 거의 없다. 그러나 인텔 기반이 아닌 시스템에서는 다음과 같은 설정이 필요하다[63].

- 각 장치에 PCI I/O와 PCI 메모리 공간을 할당한다.
- 시스템에 있는 각 PCI-PCI 브릿지의 PCI I/O와 PCI 메모리 주소 윈도우를 설정한다.
- 장치에 인터럽트 라인 값을 만든다. 이것은 그 장치의 인터럽트 처리를 제어한다.

다음 작은 절에서는 이들 코드가 어떻게 동작하는지 이야기한다.

역주 63) IBM PC외에 BIOS가 있는 시스템을 찾기 힘들며, 이런 시스템에서는 운영체제 코드가 그 역할을 맡아야 한다. (flyduck)

장치가 얼마나 많은 PCI I/O와 PCI 메모리 공간을 필요로 하는지 알아내기

찾은 PCI 장치들이 얼마나 많은 PCI I/O와 메모리 주소공간을 필요로 하는지 알아내기 위해서 이를 각 장치에게 물어야 한다. 이는 각 장치의 베이스 주소 레지스터에 전부 1을 써넣고 나서 이를 다시 읽어봄으로써 이루어진다. 장치는 자신에게 무관한 주소 비트를 0으로 설정하고, 이는 자신이 필요로 하는 주소공간의 크기를 나타내는 효과를 가지게 된다.

베이스 주소 레지스터에는 두가지 기본 유형이 있는데, 먼저 어떤 주소공간에 장치의 레지스터가 있어야 하는지를 - PCI I/O 공간인지 PCI 메모리 공간인지 - 지시한다. 이것은 레지스터의 비트 0으로서 알 수 있다. 그림 6.10에서는 PCI 메모리와 PCI I/O를 위한 베이스 주소 레지스터의 두가지 유형을 보여준다.

주어진 베이스 주소 레지스터가 얼마나 많은 주소공간을 요구하는지 알아내기 위해, 레지스터에 모두 1을 써넣고 이를 다시 읽는다. 그러면 장치는 자신에게 무관한 주소 비트를 0으로 설정하여, 필요한 주소공간의 크기를 지정하게 된다. 이런 디자인은 모든 주소공간의 크기는 2의 배수이고 이에 맞춰 자연스럽게 정렬되어 있다는 것을 알려준다.

예를 들어 DECChip 21142 PCI 이더넷 장치를 초기화할 때 장치는, PCI I/O나 PCI 메모리로 *0x100* 바이트의 공간을 필요로 한다고 알려준다. 초기화 코드는 그만큼의 공간을 할당하고, 이 순간 21142의 제어 레지스터와 상태 레지스터를 이 주소에서 볼 수 있게 된다.

PCI-PCI 브릿지와 PCI 장치에 PCI I/O와 PCI 메모리를 할당하기

다른 메모리처럼 PCI I/O와 PCI 메모리 공간도 유한하며, 어느정도 희소한 것이다. 인텔기반이 아닌 시스템에서의 PCI 확정 코드는 (그리고 인텔 시스템에서의 BIOS 코드는) 각 장치가 요구하는 크기의 메모리를 효과적인 방법으로 장치에게 할당해야 한다. PCI I/O와 PCI 메모리는 자연스럽게 정렬이 되도록 장치에 할당되어야 한다. 예를 들어, 장치가 PCI I/O 공간 *0xB0*을 요구한다면 이것은 *0xB0*의 여러배가 되는 주소에서 정렬되어야 한다는 것이다. 여기에 더해, 어떤 브릿지든지 PCI I/O와 PCI 메모리 베이스는 4K 단위로 정렬되어야 하며 각자 1MByte의 경계를 가지고 있어야 한다. 다운스트림 장치들의 주소공간이 모든 업스트림 PCI-PCI 브릿지의 메모리 공간에 위치해 있어야 하기 때문에, 공간을 효율적으로 할당하는 것은 조금 어려운 문제이다.

리눅스는 PCI 디바이스 드라이버가 만들어낸 버스/장치 트리에서 기술된 각 장치들을 주소공간에서 PCI I/O 메모리가 증가하는 쪽으로 할당하는 알고리즘을 사용한다. 여기서도 재귀적인 알고리즘을 사용하여 PCI 초기화 코드에서 만들어낸 `pci_bus`와 `pci_dev` 자료구조를 따라간다. PCI 버스의 루트부터(`pci_root`가 가리키고 있는) 시작하여 BIOS 확정 코드는 다음과 같이 하게 된다.

- 현재 있는 전역 PCI I/O와 메모리 베이스를 4K 단위로 상대적으로 1Mbyte 경계를 가지도록 정렬한다.
- 현재 버스의 모든 장치에 대해 (증가하는 쪽으로 PCI I/O 메모리를 필요로 한다)
  - 장치에게 PCI I/O와 PCI 메모리 공간을 할당하고
  - 행다하는 크기만큼 전역 PCI I/O와 메모리 베이스를 이동하고
  - 장치의 PCI I/O와 PCI 메모리를 사용하는 것을 가능하게 한다.
- 현재 버스의 모든 다운스트림 버스들을 재귀적으로 찾아 공간을 할당한다. 이는 전역 PCI I/O와 메모리 베이스를 바꾼다는 것에 주의한다.
- 현재 있는 전역 PCI I/O와 PCI 메모리 베이스를 4K 단위로 상대적으로 1Mbyte 경계로 정렬하고, 이렇게 하는 중에 현재 PCI-PCI 브릿지가 필요로 하는 PCI I/O와 PCI 메모리 윈도우의 베이스와 크기를 알아낸다.
- 이 버스에 연결된 PCI-PCI 브릿지를 프로그램하여 PCI I/O와 PCI 메모리 베이스와 크기를 지정한다.
- PCI-PCI 브릿지에 있는 PCI I/O와 PCI 메모리에 접근하도록 하는 브릿지 기능을 켠

다. 이는 브릿지의 1차 PCI 버스에서 보이는 PCI I/O와 PCI 메모리 주소 중에서 윈도우 안에 있는 PCI I/O와 PCI 메모리 주소는 2차 버스로 건너가게 된다는 것이다.

예로 들었던 그림 6.1을 생각하면, PCI 확정 코드는 다음과 같이 시스템을 설정할 것이다 :

PCI 베이스들의 정렬(Align the PCI bases) PCI I/O는 *0x4000*, PCI 메모리는 *0x100000*이다. 이것은 PCI-ISA 브릿지가 모든 주소를 ISA 주소 사이클로 변환할 수 있게 한다.

비디오 장치 이 장치는 *0x200000*만큼의 PCI 메모리를 필요로 하여, 현재 PCI 메모리 베이스 *0x200000*에서 시작하는 크기를 할당해주는데, 요구한 크기대로 자연히 정렬이 되어야 한다. PCI 메모리 베이스는 *0x400000*으로 이동하고, PCI I/O 베이스는 그대로 *0x4000*에 남아 있다.

PCI-PCI 브릿지 이제 PCI-PCI 브릿지를 건너가 거기에서 PCI 메모리를 할당한다. 여기서 베이스들이 이미 올바로 정렬이 되어 있기 때문에 정렬을 할 필요가 없다.

이더넷 장치 이것은 *0xB0* 바이트의 PCI I/O와 PCI 메모리 공간을 요구한다. 이 장치는 *0x4000*에서 시작하는 PCI I/O 공간과 *0x400000*에서 시작하는 PCI 메모리를 갖게 된다. PCI 메모리 베이스는 *0x4000B0*으로 이동하고 PCI I/O 베이스는 *0x40B0*이 된다.

SCSI 장치 이것은 *0x1000* 크기의 PCI 메모리를 요구하고, 자연 정렬이 된 후 *0x401000*에서 시작하는 메모리를 할당받는다. PCI I/O 베이스는 그대로 *0x40B0*, PCI 메모리 베이스는 *0x402000*으로 이동한다.

PCI-PCI 브릿지의 PCI I/O, PCI 메모리 윈도우 이제 브릿지로 돌아와 *0x4000*과 *0x40B0* 사이에 위치하는 PCI I/O 윈도우와, *0x400000*과 *0x402000* 사이에 위치하는 PCI 메모리 윈도우를 설정한다 이것은 PCI-PCI 브릿지가 비디오 장치에 대한 접근은 무시하고, 이더넷이나 SCSI 장치로 접근할 때에만 이를 통과시키도록 한다.

번역 : 이호
정리 : 이호

7장

인터럽트와 인터럽트 처리 (Interrupt and Interrupt Handling)

이 장에서는 리눅스 커널이 인터럽트를 어떻게 처리하는지 살펴본다. 커널이 인터럽트를 처리하는 데는 일반적인 메커니즘과 인터페이스가 있지만, 인터럽트를 처리하는 세세한 내용은 아키텍처마다 다르다.

리눅스는 서로 다른 일을 하는 수많은 하드웨어를 사용한다. 비디오 장치는 모니터를 구동하며, IDE 장치는 디스크를 구동하는 식이다. 이런 장치들은 동기적으로 구동할 수 있다, 즉 어떤 동작을 요청하고 (예를 들면 메모리 블럭을 디스크에 저장하는 것과 같은) 그것이 완료될 때까지 기다리는 것이다. 하지만 이 방법은 동작하기는 하지만 매우 비효율적이어서 운영체제는 각각의 동작이 완료될 때까지 기다려야 하므로 "아무것도 하지 않으면서 바쁜 상태(busy doing nothing)"로 많은 시간을 소비할 것이다. 이보다 더 좋고 더욱 효율적인 방법은 요청을 한 뒤 다른 더 유용한 작업을 하고 요청한 작업이 끝나면 장치로부터 인터럽트를 받는 것이다. 이런 설계를 사용하면 여러 장치에 동시에 작업을 요청하는 것이 가능하다.

CPU가 무엇을 하고있건 간에 장치가 인터럽트를 걸 수 있으려면 하드웨어에서 지원해야 한다. 모두는 아니더라도 알파 AXP와 같은 대부분의 범용 프로세서들은 대개 비슷한 방법을 사용한다. CPU의 특정한 핀의 전압이 바뀌면 (예를 들어 +5볼트에서 -5볼트로), CPU는 하던 일을 멈추고 인터럽트 처리 코드라는 인터럽트를 다루는 특별한 코드를 수행하기 시작한다. 이들 핀 중 어떤 핀은 간격 타이머에 연결되어 있어 1000분의 1초마다 인터럽트를 받으며, SCSI 컨트롤러와 같은 다른 장치에 연결된 핀들도 있을 것이다.

대체로 시스템은 인터럽트 컨트롤러를 사용하여 CPU의 인터럽트 핀으로 인터럽트를 1:1로 전달하지 않고, 장치 인터럽트를 그룹으로 묶어준다. 이렇게 하면 CPU에 있는 인터럽트 핀을 줄일 수 있을 뿐 아니라 시스템을 유연하게 디자인할 수 있다. 인터럽트 컨트롤러에는 인터럽트를 조정하는 마스크 레지스터와 상태 레지스터가 있다. 마스크 레지스터의 비트들을 켜거나 껴서 인터럽트를 가능하게 하거나 불가능하게 만들 수 있으며, 상태 레지스터는 시스템에 현재 발생한 인터럽트를 돌려준다.

시스템의 일부 인터럽트는 하드웨어적으로 연결되어 있다. 예를 들어 실시간 클럭의 간격 타이머는 영구적으로 인터럽트 컨트롤러의 3번 핀에 연결되어 있다. 그러나 어떤 핀들은 특정한 ISA 또는 PCI 슬롯에 어떤 컨트롤러 카드가 꽂혀 있는지에 따라 어떤 장치에 연결되는지 결정된다. 예를 들어 인터럽트 컨트롤러의 4번 핀이 PCI 슬롯 0번에 연결되어, 여기에 이더넷 카드를 꽂을 수도 있지만 뒤에 SCSI 컨트롤러로 바꿔 끼울 수도 있다는 것이다. 인터럽트 처리에 있어서 기본적인 사항은, 각 시스템은 독자적인 인터럽트 전달 방식을 가지고 있으며, 운영체제는 이에 대처할 수 있도록 유연하게 만들어져야 한다는 것이다.

대부분의 현대 범용 마이크로프로세서들은 인터럽트를 똑같은 방식으로 처리한다. 하드웨어

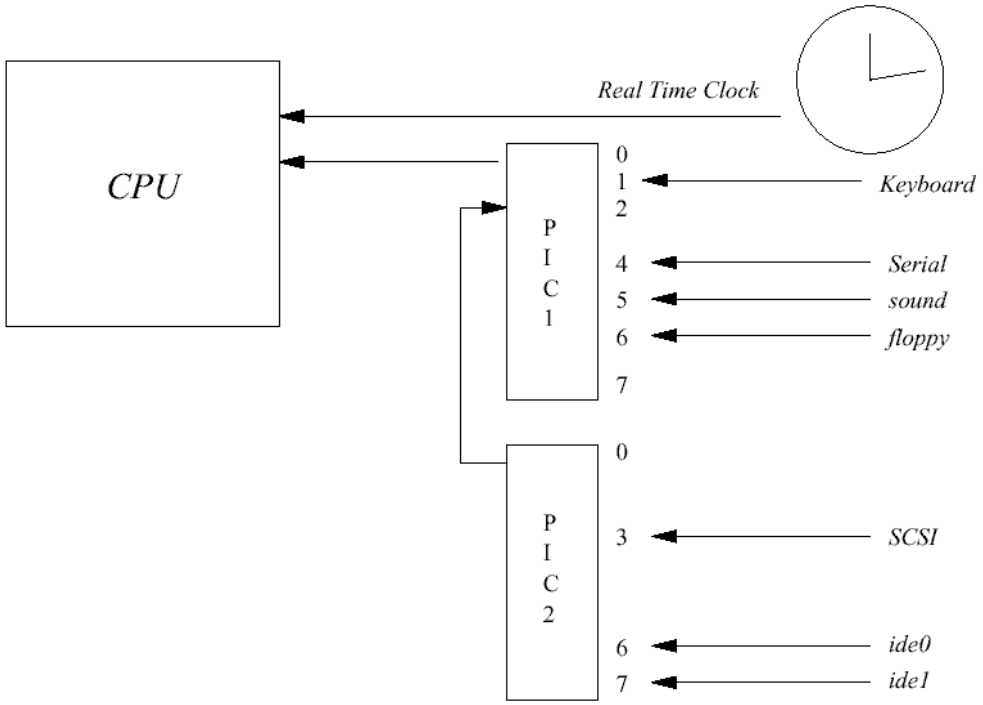


그림 7.1 : 인터럽트 전달과정에 대한 논리도

인터럽트가 발생하면 CPU는 지금 수행하고 있던 명령어의 실행을 중단하고 인터럽트 처리 코드가 있거나 인터럽트 처리 코드로 분기하는 명령어가 있는 메모리 번지로 점프한다. 이 코드들은 일반적으로 인터럽트 모드(interrupt mode)라고 하는 CPU의 특별한 모드에서 수행되는데, 보통 이 모드에서는 다른 인터럽트가 발생할 수 없다. 물론 예외가 있다. 어떤 CPU에서는 인터럽트에 우선순위를 매겨 더 높은 우선순위의 인터럽트가 발생할 수 있게 한다. 이런 경우 가장 높은 순위의 인터럽트 처리 코드는 아주 주의해서 작성해야 하며, 종종 자신의 스택을 가지고 있어 인터럽트를 처리하기 전에 여기에 CPU의 수행상태(즉, CPU의 일반 레지스터와 컨텍스트 모두)를 저장하는데 사용한다. 어떤 CPU에는 인터럽트 모드에서만 존재하는 특별한 레지스터 세트가 있어, 인터럽트 코드는 필요한 컨텍스트를 저장하는데 이 레지스터들을 사용할 수 있다.

인터럽트가 처리되고 나면 CPU의 상태는 인터럽트 이전으로 복구되고 인터럽트는 해제된다. 그러면 CPU는 인터럽트가 발생하기 전에 수행하던 것을 계속 실행하게 된다. 중요한 것은 인터럽트를 처리하는 코드는 가능한 효율적이어야 하며 운영체제는 인터럽트를 너무 자주 또는 너무 오래 막고 있지 않아야 한다는 점이다.

7.1 프로그램 가능 인터럽트 컨트롤러(Programmable Interrupt Controller, PIC)

시스템 디자이너는 자신이 원하는 어떤 인터럽트 구조라도 사용할 수 있지만, IBM PC는 인텔 82C59A-2 CMOS PIC [6, 인텔 주변 장치]나 그 유사형을 사용한다. 이 컨트롤러는 PC의 초창기때부터 널리 사용된 것으로, ISA 주소공간에 있는 컨트롤러의 레지스터를 이용해 (이 레지스터의 위치는 고정되어 이미 알려져 있다) 프로그래밍을 할 수 있다. 가장 최근의 로직 칩 세트도 ISA 메모리의 같은 위치에 동등한 레지스터를 가지고 있다. 알파 AXP 기반 PC와 같이 인텔에 기반하지 않은 시스템들은 이러한 구조적 제약으로부터 자유로우며, 대개 다른 인터럽트 컨트롤러를 사용한다.

그림 7.1에서 8비트 컨트롤러 PIC1과 PIC2가 같이 연결되어 있으며, 각각 마스크 레지스터

와 인터럽트 상태 레지스터 하나씩을 가지고 있는 것을 볼 수 있다. 마스크 레지스터는 주소 *0x21*과 *0xA1*에 있으며 상태 레지스터는 *0x20*과 *0xA0*에 있다. 마스크 레지스터의 특정한 비트에 1을 쓰면 해당 인터럽트를 가능하게 하며, 0을 쓰면 인터럽트를 불가능하게 한다. 즉, 세번째 비트에 1을 쓰면 인터럽트 3번을 가능하게 하는 것이며, 0을 쓰면 불가능하게 하는 것이다. 불행하게도 (또한 귀찮게도), 인터럽트 마스크 레지스터는 쓸 수만 있으며, 거기에 써 놓은 값을 읽어올 수는 없다. 따라서 리눅스는 마스크 레지스터에 어떤 것을 설정하였는지를 따로 복사하여 보관하여야만 한다. 리눅스는 인터럽트 허용 루틴과 인터럽트 금지 루틴에서, 이 보관된 마스크를 변경하고 매번 레지스터에 전체 마스크를 쓴다.

인터럽트가 발생하면, 인터럽트 처리 코드는 두 인터럽트 상태 레지스터(Interrupt Status Register, ISR)를 읽는다. 인터럽트 처리 루틴은 *0x20*에 있는 ISR을 16비트 인터럽트 레지스터의 하위 여덟 비트로, *0xA0*에 있는 ISR을 상위 여덟 비트로 처리한다. 따라서 *0xA0*에 있는 ISR의 첫번째 비트에 해당하는 인터럽트는 시스템 인터럽트 9로 취급하게 된다. PIC1에 있는 두번째 비트는 PIC2에서 발생하는 인터럽트를 연결하는데 사용하기 때문에 사용할 수 없다. PIC2에 발생하는 어떤 인터럽트든지 PIC1의 두번째 비트를 설정하게 된다.

7.2 인터럽트 처리용 자료구조의 초기화

커널의 인터럽트 처리용 자료구조는 디바이스 드라이버들이 시스템의 인터럽트에 대한 제어권을 요청하면서 셋업된다. 이를 위해 디바이스 드라이버는 일련의 리눅스 커널 서비스를 사용함으로써, 인터럽트를 요청하고, 인터럽트를 가능하게 하거나, 불가능하게 만든다. 개별 디바이스 드라이버는 이런 루틴들을 불러 자신의 인터럽트 처리 루틴의 주소를 등록한다.

arch/*/kernel/irq.c 에 있는 request_irq(), enable_irq(), disable_irq() 참조

PC 아키텍처의 관례상 몇몇 인터럽트들은 (특정한 장치가 사용하도록) 지정되어 있는데, 이 경우에 해당 디바이스 드라이버는 초기화될 때 간단하게 그 지정된 인터럽트를 요청하면 된다. 플로피 디스크 디바이스 드라이버가 이와 같이 동작하는데, 항상 IRQ 6번을 요청한다. 하지만 장치가 어떤 인터럽트를 사용하게 될 것인지 디바이스 드라이버가 모르는 경우도 있다. PCI 디바이스 드라이버의 경우에는 장치가 어떤 인터럽트를 사용하는지 항상 알고 있기 때문에 문제가 되지 않지만, 불행하게도 ISA 디바이스 드라이버의 경우에는 자신이 사용할 인터럽트 번호를 쉽게 찾을수 있는 방법이 없다[64]. 리눅스에서는 이런 문제를 해결하기 위해 디바이스 드라이버가 자신이 사용할 인터럽트를 탐사(probe)할 수 있도록 허용하고 있다.

arch/*/kernel/irq.c 에 있는 irq_probe_*() 참조

먼저 디바이스 드라이버는 장치에 무엇인가를 해서 인터럽트가 발생하도록 한다. 그런 후 다른 장치에 할당되지 않은 시스템의 모든 인터럽트들을 가능하게 한다. 이렇게 하면 처음에 발생시켰던 장치의 인터럽트가 PIC를 통해 전달될 것이다. 리눅스는 인터럽트 상태 레지스터를 읽어 그 내용을 디바이스 드라이버에게 돌려준다. 이 값이 0이 아니라면 탐사 중에 하나 이상의 인터럽트가 발생한 것이다. 드라이버는 탐사를 종료하고 다른 장치에 할당되지 않은 인터럽트를 모두 불가능하게 한다[65]. 탐사를 통해 ISA 디바이스 드라이버가 자신이 사용할 IRQ 번호를 찾았다면, 정상적으로 이에 대한 통제권을 요청할 수 있다.

ISA 기반 시스템에 비해 PCI 기반 시스템은 훨씬 더 동적이다. ISA 장치가 사용하는 인터럽트 핀은 대개 하드웨어 장치 위에 있는 점퍼를 사용해 설정하고, 디바이스 드라이버에 이 값이 지정되어 있다. 반면에, PCI 장치가 사용할 인터럽트는 시스템이 부팅하면서 PCI를 초

역주 64) ISA 장치는 인터럽트를 하드웨어에 있는 점퍼로 설정하거나 EPROM에 값을 기록함으로써 지정하도록 되어 있다. 하지만 이들 값을 운영체제에서 알아낼 수 있는 방법은 없다. (flyduck)

역주 65) 커널에는 이를 위해 `probe_irq_on()`과 `probe_irq_off()` 함수가 있다. 앞의 함수를 불러 할당되지 않은 인터럽트를 가능하게 한 후, 인터럽트를 발생한 후, 뒤의 함수를 부르면 발생한 인터럽트를 돌려주고, 인터럽트 상태를 원상태로 복구한다. `probe_irq_off()` 함수는 인터럽트가 발생하지 않으면 0을, 하나의 인터럽트가 발생하면 해당 인터럽트 번호를, 둘 이상의 인터럽트가 발생하여 모호한 경우 음수를 돌려준다. (flyduck)

기화 할 때 PCI BIOS나 PCI 서브시스템이 할당해 준다. 각각의 PCI 장치는 A, B, C, D의 4개의 인터럽트 핀을 사용할 수 있다. 어떤 핀을 사용할 지는 장치를 만들 때 결정되는데, 대부분은 기본적으로 A 핀에 있는 인터럽트로 설정한다. 각 PCI 슬롯에 있는 PCI 인터럽트 라인 A, B, C, D는 인터럽트 컨트롤러에 연결되어 있다. 예를 들어 PCI 슬롯 4의 A 핀은 인터럽트 컨트롤러의 6번 핀에 연결하고, PCI 슬롯 4의 B 핀은 인터럽트 컨트롤러의 7번 핀에 연결하는 식으로 되어 있다.

arch/alpha/kernel/bios32.c 참조

PCI 인터럽트가 어떻게 전달되는지는 시스템마다 다르므로, PCI 인터럽트 전달 구조를 이해할 수 있는 셋업 코드가 필요하다. 인텔 칩을 사용하는 PC에서는 시스템이 부팅할 때 실행되는 시스템 BIOS가 이 역할을 하는데, 알파 AXP를 사용하는 시스템과 같이 BIOS가 없는 시스템의 경우에는 리눅스 커널이 이러한 설정을 한다. PCI 셋업 코드는 각 장치별로 인터럽트 컨트롤러의 핀 번호를 PCI 설정 헤더에 쓴다. 그리고 장치가 사용하는 PCI 슬롯 번호와 PCI 인터럽트 핀 번호 및 PCI 인터럽트 전달 구조를 이용하여 인터럽트 핀 (또는 IRQ) 번호를 결정한다. 이러한 방법으로 장치가 사용할 인터럽트 핀 번호가 고정되고, 인터럽트 핀 번호는 이 장치를 관리하는 PCI 설정 헤더에 있는 항목에 저장된다. 셋업 코드는 이 정보를 이러한 목적으로 마련된 인터럽트 라인 항목에 적어 넣는다. 디바이스 드라이버는 이 정보를 읽어다 리눅스 커널에게 인터럽트에 대한 제어권을 요청할 때 사용한다.

PCI-PCI 브릿지를 사용할 때와 같이 시스템에 PCI 인터럽트를 일으키는 장치가 많은 경우가 있다. 인터럽트를 일으키는 장치가 시스템의 PIC에 있는 핀 수보다 많을 수 있다. 이 경우 PCI 장치라면, 인터럽트를 공유하여 여러 PCI 장치의 인터럽트가 인터럽트 컨트롤러의 핀 하나에 발생하게 할 수 있다[66]. 이런 인터럽트 공유를 지원하기 위해 리눅스는 해당 인터럽트의 제어권을 처음으로 요청하는 디바이스 드라이버가 인터럽트를 공유할 수 있는지를 밝히도록 하고 있다[67]. 인터럽트를 공유하기 위해 `irq_action` 벡터에 `irqaction` 자료구조를 여러 개 담게 된다. 공유 인터럽트가 발생하면, 리눅스는 그 인터럽트를 사용하는 장치의 인터럽트 핸들러를 모두 불러준다. 인터럽트를 공유할 수 있는 모든 디바이스 드라이버(모든 PCI 디바이스 드라이버겠지만)는 서비스할 인터럽트가 없는 경우라 하더라도 인터럽트 핸들러가 불릴 수 있으므로 이에 대비해야 한다[68].

7.3 인터럽트 처리

리눅스의 인터럽트 처리 서브시스템의 주요한 임무중 하나는 인터럽트를 올바른 인터럽트 처리 코드로 전달하는 것이다. 따라서 인터럽트 처리 서브시스템은 시스템의 인터럽트 전달 구조를 파악하고 있어야만 한다. 예를 들어 플로피 컨트롤러가 인터럽트 컨트롤러의 6번 핀에 인터럽트를 일으킨다면[69], 리눅스 커널의 인터럽트 처리 서브시스템은 이 인터럽트가 플

역주 66) 사실 PCI에서는 규약으로 인터럽트 공유가 가능하도록 되어 있지만, ISA라고 해서 인터럽트를 공유할 수 없는 것은 아니다. 물론 ISA 규약에는 인터럽트 공유에 대한 규정이 없고 초창기에 나온 카드는 전기적인 문제로 인터럽트 공유에 문제가 있었지만 지금 있는 대부분의 ISA 카드는 하드웨어적으로 인터럽트 공유에 문제가 없다. 따라서 인터럽트 공유의 문제는 대부분 소프트웨어 문제이며, 리눅스는 ISA 디바이스 드라이버라고 하더라도 인터럽트 핸들러를 등록할 때 인터럽트를 공유할 수 있는지 지정할 수 있다. (flyduck)

역주 67) 물론 인터럽트를 인터럽트를 공유하지 않는 인터럽트 핸들러가 설치되어 있다면 인터럽트를 공유하는 핸들러를 등록할 수 없을 것이며, 반대의 경우도 마찬가지다. (flyduck)

역주 68) 즉 자신이 처리하는 장치에서 인터럽트가 발생하지 않았더라도 인터럽트를 공유하는 다른 장치에서 발생한 인터럽트 때문에 자신의 인터럽트 핸들러가 불릴 수 있다는 것이다. 이는 자신이 제어하는 장치에 있는 인터럽트 상태 레지스터를 읽어서 인터럽트가 발생한 경우 이를 처리하고, 그렇지 않은 경우에는 그냥 무시하면 된다. 그러면 실제로 인터럽트가 발생한 장치의 디바이스 드라이버가 이를 처리할 것이다. (flyduck)

69) 사실 플로피 컨트롤러는 관례상 PC 시스템에서 인터럽트가 고정된 장치 중 하나이다. 플로피 컨트롤러는 항상 인터럽트 6번에 연결된다.

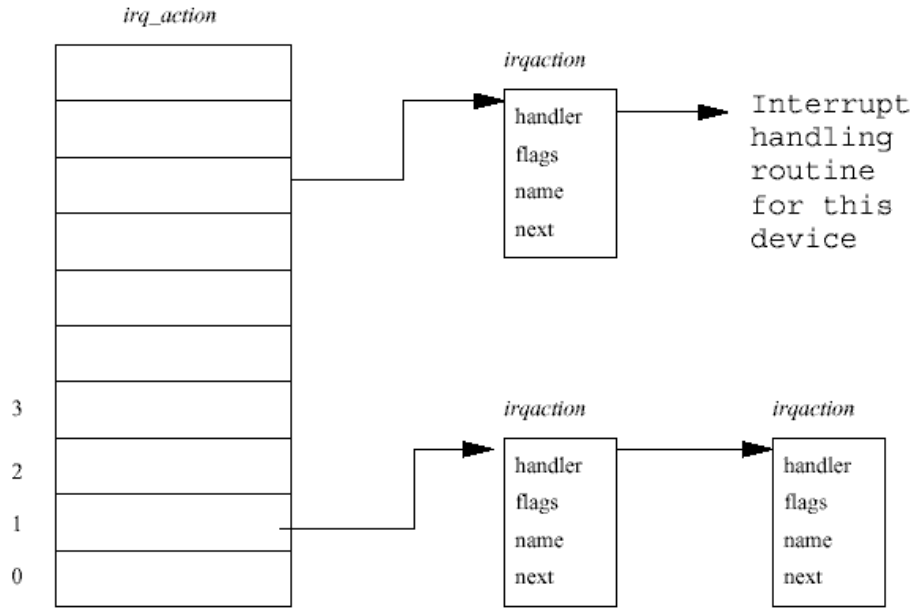


그림 7.2 : 리눅스 인터럽트 처리 자료구조

로피에서 발생할 것임을 인지하고 이것을 플로피 디바이스 드라이버의 인터럽트 처리 코드로 전달해야 한다. 이를 위해 리눅스는 시스템의 인터럽트를 처리하는 루틴들의 주소를 가지고 있는 자료구조에 대한 일련의 포인터를 사용한다. 이 루틴들은 해당 디바이스 드라이버에 있는 것이며, 드라이버가 초기화될 때 자신이 사용할 인터럽트를 요청하는 것은 각 드바이스 드라이버의 책임이다. 그림 7.2는 `irqaction` 자료 구조를 가리키고 있는 포인터들의 벡터인 `irq_action`을 보여주고 있다. 각 `irqaction` 자료구조는 인터럽트 처리 루틴의 주소를 포함해 해당 인터럽트에 대한 핸들러 정보를 담고 있다. 인터럽트의 수와 이들이 어떻게 처리되는지는 아키텍처마다, 때때로 시스템마다 다르기 때문에 리눅스의 인터럽트 처리 코드는 아키텍처 종속적이다. 즉, `irq_action` 벡터의 크기는 시스템에 있을 수 있는 인터럽트를 일으키는 장치의 숫자에 따라 달라진다.

인터럽트가 발생하면, 리눅스는 먼저 시스템에 있는 PIC의 인터럽트 상태 레지스터(ISR)를 읽어 어느 장치가 인터럽트가 일으켰는지 알아낸다. 그런 후 리눅스는 그 장치를 `irq_action` 벡터의 오프셋으로 변환한다. 예를 들어, 플로피 컨트롤러가 일으키는 인터럽트 컨트롤러 6번 핀에 발생한 인터럽트는 인터럽트 핸들러 벡터의 일곱번째 포인터로 변환된다. 인터럽트가 발생하였는데 이를 처리할 인터럽트 핸들러가 없다면 리눅스 커널은 오류를 기록할 것이다. 핸들러가 있다면 이 인터럽트를 일으키는 모든 장치에 대해 `irqaction` 자료구조에 있는 인터럽트 처리 루틴을 부를 것이다.

리눅스 커널이 디바이스 드라이버의 인터럽트 처리 루틴을 부르면, 이 루틴은 왜 인터럽트가 발생하였지를 파악하여 이에 효율적으로 반응해야 한다. 왜 인터럽트가 발생하였지를 파악하기 위해 디바이스 드라이버는 인터럽트가 발생한 장치의 상태 레지스터를 읽을 수 있을 것이다. 그 이유는 오류였을 수도 있고 요청한 작업이 완료됐다고 보고한 것일 수도 있다. 예를 들어 플로피 컨트롤러는 플로피 디스크의 정확한 섹터 위에 플로피의 헤드를 올려 놓았다고 보고할 수 있다. 인터럽트가 발생한 이유를 알아 냈다면, 디바이스 드라이버는 인터럽트를 처리하기 위해 더 많은 작업을 해야 할 필요가 있을 있다. 그런 경우 리눅스 커널에는 디바이스 드라이버가 그 작업을 뒤로 연기할 수 있는 메커니즘이 있다[70]. 이것은 CPU가 너무 오래동안 인터럽트 모드에 있는 것을 피하려는 것이다[71]. 더욱 자세한 내용은 디바이스

역주 70) 이런 방법으로 하반부 핸들러(bottom half handler)와 작업큐(task queue)가 있다. (flyduck)

역주 71) 인터럽트를 처리하는 도중에는 다른 인터럽트를 발생하지 못하도록 하기 때문에

드라이버 장(8장)을 보라.

REVIEW NOTE : Fast interrupt와 slow interrupt는 인텔에 있는 개념인가?[72]

번역 : 김성룡, 홍경선
정리 : 이호

(모든 인터럽트이든, 우선순위가 낮은 인터럽트이든), 인터럽트 처리 상태에 오래 있는 것은 좋지 않다. (flyduck)

역주 72) 빠른 인터럽트(fast interrupt)와 느린 인터럽트(slow interrupt)는 인터럽트 처리 방식의 차이이다. 빠른 인터럽트는 인터럽트 처리가 원자적으로(atomic) 이루어지는 경우이고, 느린 인터럽트는 그렇지 않다. 실질적인 차이는 빠른 인터럽트의 경우 프로세서에서 인터럽트를 금지시켜 처리중 다른 인터럽트의 방해를 받지 않지만, 느린 인터럽트는 다른 인터럽트에 의해 중지될 수 있다. 그리고 빠른 인터럽트는 인터럽트 핸들러는 앞뒤에 하는 일이 적이 보다 빠르다. 알파나 Sparc에서는 이런 차이는 없으며, 인텔에서도 2.1.37 버전 이후에 이 차이는 없어졌다. (flyduck)

8 장

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 장치에 들었있든 상관없이, 마운트된 파일 시스템들을 일관된 모습으로 보여준다. 이 장에서는 리눅스 커널이 시스템에 있는 물리적인 장치를 어떻게 관리하는지 살펴보기로 한다.

CPU 가 시스템에 있는 지능을 가진 유일한 장치는 아니다. CPU 말고도 모든 물리적 장치들은 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 CSRs 과 NCR 810 SCSI 컨트롤러의 CSRs 는 완전히 다른다. CSRs 는 장치를 시작하고 멈추고, 초기화하며 문제가 발생했을 때 이를 진단하는데 이용된다. 모든 응용프로그램에 하드웨어를 관리하는 코드를 넣지 않으며, 리눅스 커널만 그 코드를 가지고 있다. 하드웨어 컨트롤러를 다루고 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 하드웨어 장치들은 보통 파일처럼 보이며, 파일을 다루는 데 쓰이는 표준 시스템 콜과 똑같은 함수를 이용하여 열고, 닫고, 읽고, 쓸 수 있다[73]. 시스템의 모든 장치는 장치 특수 파일(device special `file`)로 표시가 된다. 예를 들어, 시스템에 있는 첫번째 IDE 디스크는 `/dev/hda` 로 나타낸다. 블럭 (디스크) 장치(block device)나 문자 장치(character device)를 나타내는 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[74]. 똑같은 디바이스 드라이버로 제어되는 모든 장치는 똑같은 메이저 장치 번호를 갖는다. 마이너 장치 번호는 다

역주 73) 이렇게 장치를 파일로 표시하는 것은 Windows 운영체제에도 영향을 미쳐, Windows 95에서는 디바이스 드라이버를 파일로 접근할 수 있으며, Windows NT 계열에서는 유닉스와 보다 가깝게 디바이스 드라이버가 장치 파일을 만드는 형태로 되어 있다. (flyduck)

역주 74) 즉 보통 문자 장치나 블럭 장치는 실제로 장치가 존재하지 않더라도 장치 특수 파일이 존재한다. 이는 실제 시스템에 장치가 많지 않더라도, `/dev` 디렉토리에 수많은 장치 파일이 존재하는 이유이다. 하지만 네트웍 장치 파일은 실제로 장치가 존재하는 경우에만 만들어진다. 예를 들어 시스템에 이더넷 장치가 있어야 `/dev/eth0`이라는 장치 특수 파일이 생긴다. (flyduck)

른 장치나 컨트롤러를 구분하는데 사용한다[75]. 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 2 를 갖는다. 블럭 장치에 있는 파일 시스템을 마운트하는 경우처럼 시스템 콜에 장치 특수 파일을 전달하면, 리눅스는 메이저 장치 번호와 여러 시스템 테이블을 이용하여(이런 것 중의 하나로 문자 장치 테이블인 chrdevs 가 있다), 장치 특수 파일을 장치의 디바이스 드라이버로 연결한다.

fs/devices.c 참조

리눅스는 문자, 블럭, 네트웍, 이 세가지 종류의 하드웨어 장치를 지원한다. 문자 장치는 버퍼를 통하지 하고 바로 읽고 쓸 수 있는 장치로, /dev/cua0 과 /dev/cua1 같은 직렬 포트가 여기에 속한다. 블럭 장치는 일정한 블럭 크기(보통 512 또는 1024 바이트이다)의 배수로만 읽고 쓸 수 있다. 블럭 장치는 버퍼 캐시(buffer cache)를 통해서 읽고 쓰며, 아무 곳이나 접근할 수 있다. 즉 어떤 블럭이든 그것이 장치의 어디에 있든지 간에 읽고 쓸 수 있다는 것이다. 블럭 장치는 장치 특수 파일을 통해서 접근할 수도 있지만, 보통은 파일 시스템을 통해서 접근한다. 블럭 장치만이 마운트되는 파일 시스템을 지원할 수 있다. 네트웍 장치는 BSD 소켓 인터페이스로 접근하며, 이는 10 장에 있는 네트워킹 서브시스템 부분에서 자세히 이야기한다.

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

커널 코드 디바이스 드라이버는 커널의 한 부분이므로, 커널의 다른 코드와 마찬가지로 잘못되면 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 인터페이스를 제공해야 한다. 예를 들어, 터미널 드라이버는 리눅스 커널에 파일 I/O 인터페이스를 제공해야 하며, SCSI 디바이스 드라이버는 커널에 파일 I/O 와 버퍼 캐시 인터페이스를 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 더이상 필요하지 않을 때 언로드할 수 있다. 이는 커널을 매우 융통성 있게 만들고 시스템의 자원을 효율적으로 이용할 수 있게 한다[76].

역주 75) 즉 메이저 장치 번호는 디바이스 드라이버에게 부여되는 것이다. 그러므로 서로 다른 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 드라이버를 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 지원한다는 것은 아니다. 이는 SCSI 클래스 디바이스 드라이버가 있어서 이것이 실제로 디바이스 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 76) 이 특성은 현재 커널이 지원하지 않는 장치가 추가되었더라도, 커널을 새로 컴파일하지 않고 해당하는 디바이스 드라이버를 추가하여 로드함으로써 장치를 사용할 수 있게 한다. (flyduck)

설정가능(Configurable) 리눅스 디바이스 드라이버를 커널에 포함하여 컴파일 할 수 있다. 어떤 장치를 넣을 것인지는 커널을 컴파일할 때 설정할 수 있다[77].

동적(Dynamic) 시스템이 부팅하고 디바이스 드라이버가 초기화 될 때, 시스템은 자신이 제어할 수 있는 하드웨어 장치를 찾는다. 만약 어떤 디바이스 드라이버가 제어할 수 있는 장치가 없다고 하더라도 문제가 안된다. 이 경우 디바이스 드라이버는 단지 여분으로 있는 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 42 번 섹터를 읽어라"), 디바이스 드라이버는 그 명령이 언제 끝났는지 아는 방법을 선택할 수 있다. 디바이스 드라이버는 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 드라이버는 커널의 한 부분이기 때문에, 만약 드라이버가 폴링만 하려고 한다면 장치가 작업을 끝마칠 때까지 커널의 다른 부분이 수행될 수 없으므로 끔찍한 일이 벌어질 것이다. 그래서 폴링을 하는 디바이스 드라이버는 시스템 타이머를 이용하여 어느정도 시간이 지나면 커널이 디바이스 드라이버에 있는 한 루틴을 부르도록 한다. 그러면 이 타이머 루틴은 명령이 수행되었는지 상태를 검사한다[78]. 이는 리눅스의 플로피 드라이버에서 사용하는 방법이다. 타이머를 이용하는 폴링은 좋은 방법이지만, 이보다 더 효과적인 방법으로 인터럽트를 사용하는 것이 있다.

제어하는 하드웨어 장치가 서비스를 받아야 할 때 하드웨어 인터럽트를 발생하는 것이 인터럽트를 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 루틴의 주소와 자신이 사용하고 싶은 인터럽트 번호를 커널에 등록한다. 현재 디바이스 드라이버가 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[79]. 시스템의 어떤 인터럽트들

역주 77) 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 78) 이는 11.3 장에서 설명하고 있는 타이머 메커니즘이다. (flyduck)

역주 79) 인터럽트 자원을 요청하는 것은 꼭 드라이버 초기화 때가 아니라도 할 수 있다. 사람에 따라서 드라이버 초기화 때에는 어떤 인터럽트를 사용하고 있는지 확인만 하고, 실제 인터럽트를 요청하는 것은 장치를 사용할 때에만 하며, 사용하지 않을 때는 인터럽트 자원을 반납하는 것이 좋다고 하는 사람도 있다. (flyduck)

은 처음부터 고정되어 있는데, 이는 IBM PC 구조의 오랜 유물이다. 그래서 플로피 컨트롤러는 언제나 인터럽트 6 을 사용한다. 다른 인터럽트들, 예를 들어 PCI 장치에서 발생하는 인터럽트들은 부팅시에 동적으로 할당된다. 이경우 디바이스 드라이버는 인터럽트의 소유권을 요청하기 이전에 자신이 제어할 장치의 인터럽트 번호 (IRQ)를 먼저 알아내야 한다. 리눅스는 PCI 에서 사용하는 인터럽트에 대해, IRQ 번호를 포함하여 시스템에 있는 장치 정보를 알 수 있는 표준 PCI BIOS 콜백을 지원한다.

인터럽트가 CPU 에 어떻게 전달되는지는 하드웨어 구조에 따라 다르지만, 대부분 구조에서는 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 장치에서 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 이용할 뿐이다. 그러나 하드디스크 컨트롤러나 이더넷 장치같이 고속도 장치들의 데이터 전송률은 매우 높다. SCSI 장치는 1 초에 40MB 까지 데이터를 전송할 수 있다.

DMA 는 이런 문제를 해결하기 위해 개발되었다. DMA 컨트롤러는 CPU 가 개입하지 않고 장치와 시스템의 메모리 사이에 데이터를 전송할 수 있도록 한다. PC 의 ISA DMA 컨트롤러는 여덟개의 DMA 의 채널을 가지고 있으며, 이 중 7 개를 디바이스 드라이버가 사용할 수있다. 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 DMA 를 시작해도 좋다고 명령한다. 데이터 전송이 완료되면 장치는 PC 에 인터럽트를 발생한다. 전송이 이루어지는 동안에 CPU 는 다른일을 맘대로 할 수 있다.

디바이스 드라이버는 DMA 를 매우 조심해서 사용해야 한다. 무엇보다도 DMA 컨트롤러는 가상 메모리에 대해서 아무것도 모르고 있으며, 그저 시스템의 물리적 메모리에 접근할 뿐이다. 따라서 DMA 에서 사용하는 메모리는 물리적인 메모리에서 연속된 블럭으로 되어 있어야 한다. 이는 프로세스의 가상 메모리 주소공간으로 DMA 를 바로 사용할 수 없다는 말이다[80]. 어쨌든 사용자는 프로세스의 물리적 페이지를 메모리에 락(lock)을 걸어[81], DMA 작업 중에 메모리가 스왑 장치로 스왑 아웃되는 것을 방지하게 만들수 있다. 둘째로, DMA 컨트롤러는 물리적 메모리 전체에 접근할 수 없다. DMA 채널의 주소 레지스터는 DMA 어드레스의 처음 16 bit 를 나타내고, 페이지 레지스터에 다음 8 비트가 있다. 즉 DMA 가 사용할 수 있는 메모리는 하부 16MB 로 제한되어 있다는 것이다.

DMA 채널은 오직 7 개 밖에 사용할 수 없고, 디바이스 드라이버들이 같이 공유할 수 없는

역주 80) 프로세스는 가상 메모리를 사용하므로 할당받은 메모리는 가상 메모리 상에서는 연속되어 있더라도 물리적으로 연속된 것은 아니다. 그래서 리눅스 커널은 DMA를 위해 특별한 메모리 할당 함수를 제공한다. (flyduck)

역주 81) 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 존재하게 하고 스왑 아웃되지 않게 한다는 의미이다. (flyduck)

드문 자원이다. 인터럽트와 마찬가지로 디바이스 드라이버는 어떤 DMA 채널을 사용할 지를 알아야 한다. 역시 인터럽트에서처럼 어떤 장치가 사용하는 DMA 채널은 고정되어 있다. 예를 들어, 플로피 장치는 항상 DMA 채널 2 번을 사용한다. 가끔은 장치가 사용하는 DMA 채널은 점퍼로 설정할 수 있다. 많은 이더넷 장치들은 이런 기술을 사용한다. 이보다 더 융통성 있는 장치들은 어떤 DMA 를 채널을 사용할 것인지 알려줄 수 있어서 (자신의 CSRs 을 통하여), 디바이스 드라이버는 단지 비어있는 DMA 채널을 사용하면 된다.

리눅스는 DMA 채널 하나당 있는 `dma_chan` 자료구조의 벡터를 이용하여 DMA 채널의 사용여부를 추적할 수 있다. `dma_chan` 자료구조는 두개의 항목으로 되어 있는데, 하나는 DMA 채널의 소유자를 나타내는 문자열이고, 다른 하나는 DMA 채널이 할당되어 있는지 비어 있는지를 나타내는 플래그이다. `cat /proc/dma` 라는 명령을 내리면 나오는 것이 이 `dma_chan` 자료구조의 벡터이다.

8.3 메모리

디바이스 드라이버는 메모리를 사용할 때 주의해야 한다. 디바이스 드라이버는 리눅스 커널의 일부분이므로 가상 메모리를 사용할 수 없다. 디바이스 드라이버가 실행될 때, 즉 인터럽트를 받았다던지 하반부 핸들러(bottom half handler)나 작업큐 핸들러(task queue handler)가 스케쥴되었을 때, `current` 프로세스는 바뀔 수 있다[82]. 디바이스 드라이버는 특정한 프로세스가 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 드라이버의 코드의 일부로 포함될 수도 있지만, 이는 커널을 필요이상으로 크게 만들어 낭비적이다. 대부분의 디바이스 드라이버는 커널의 페이지되지 않는 메모리(non-paged)를 할당받아 자신의 자료를 넣는다.

리눅스는 커널 메모리를 할당하고 해제하는 루틴을 제공하는데, 디바이스 드라이버는 이를 사용한다. 커널 메모리는 2 의 제곱승 단위로 할당된다. 예를 들면 128 이나 512 크기로 할당되는데, 디바이스 드라이버가 더 작은 크기를 요청해도 이렇게 할당된다. 디바이스 드라이버가 요청하는 크기는 다음 블럭의 크기에 맞춰 올림하여 할당된다. 이렇게 하면 프리 블럭들을 합쳐 더 큰 블럭을 만들 수 있으므로, 커널 메모리 해제가 쉬워진다[83].

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 발생하는 것은 원하지 않으며, 이 경우 곧바로 메모리를 할당할 수 없다면 커널 메모리 할당 루틴은 실패할 수도 있다. 만약 디바이스 드라이버가 할당받은 메모리를 DMA 로 입출력을 하기를 원한다면, 그 메모리를 요구할 때 DMA 가능이라고 지정할 수 있다. 이렇게 한 시스템에 DMA 가능 메모리를 구성하는 것을 알아야 하는 것은 리눅스 커널이지 디바이스 드라이버가 아니다.

역주 82) 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_struct`를 가리키고 있을 때) 실행되지만, 인터럽트 핸들러나 하반부 핸들러, 작업큐로 처리될 때는 현재 프로세스는 전혀 상관없는 프로세스 일 수 있다 (작업큐에서 `tq_scheduler`는 인터럽트 타임에서 처리되지 않는다). 그래서 이들 처리 루틴에서는 현재 프로세스에 의존할 수 없다. 리눅스 커널에서 `current`는 현재 프로세스의 `task_struct`를 가리킨다. (flyduck)

역주 83) 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)

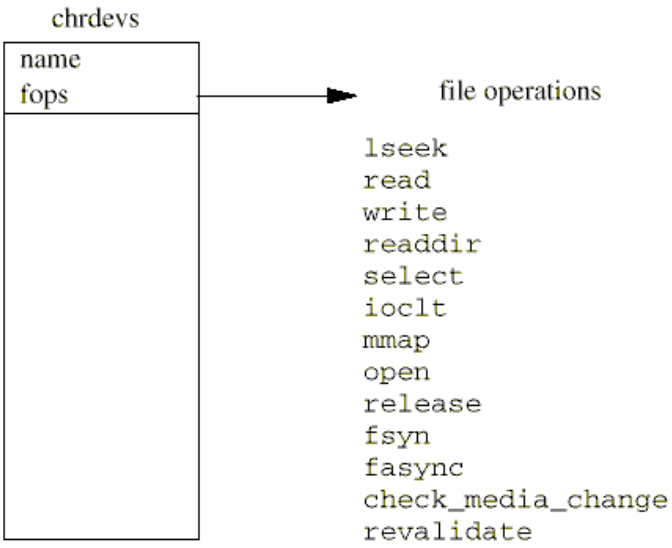


그림 8.1 : 문자 장치

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 이들로에게 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 SCSI 와 IDE 디스크는 매우 다르게 동작하지만, 리눅스 커널은 똑같은 인터페이스를 통해 이들을 사용한다.

리눅스는 매우 동적이다. 리눅스 커널은 부팅할 때마다 다른 물리적 장치들을 알게 되고, 다른 디바이스 드라이버를 필요로 하게 된다. 리눅스는 커널을 빌드할 때 설정 스크립트를 통하여 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 드라이버의 동적인 성격을 원할하게 하기 위해, 디바이스 드라이버는 자신이 초기화될 때 커널에 자기 자신을 등록한다. 리눅스는 디바이스 드라이버와의 인터페이스의 한 부분으로서, 등록된 디바이스 드라이버의 테이블을 관리한다. 이들 테이블은 해당하는 종류의 장치와 인터페이스를 제공하는 함수들의 포인터와 정보를 가지고 있다.

8.4.1 문자 장치(Character Device)

include/linux/major.h 참조

문자 장치는 리눅스의 장치들 중에서 가장 단순한 것이다. 프로그램은 그 장치가 마치 파일인 것처럼 표준 시스템 콜을 사용하여 열고, 읽고, 쓰고, 닫을 수 있다. 이러한 사실은 그 장치가 PPP 데몬이 리눅스 시스템을 인터넷에 연결하기 위해 사용하는 모뎀이라 할 지라도 마찬가지다. 문자 장치가 초기화 될 때 이것의 디바이스 드라이버는, `device_struct` 자료구조의 벡터인 `chrdevs` 에 자신의 엔트리를 추가함으로써 리눅스 커널에 자신을 등록한다[84]. 장치의 메이저 장치 번호는 (예를 들어 `tty` 장치에 할당되는 4 번) 이들 배열의 인덱스로서 사용된다. 장치에 대한 메이저 장치 번호는 고정되어 있다. `chrdevs` 벡터의 각 원소인

역주 84) 이러한 일을 하는 시스템 콜은 `register_chrdev()`로, 여기에는 장치의 메이저 번호와 디바이스 드라이버의 이름, 그리고 파일 연산 블럭이 전달된다. include/linux/fs.h에서 함수의 프로토타입(prototype)을 볼 수 있다.(flyduck)

device_struct 자료구조는 두가지 항목을 가지고 있다. 하나는 디바이스 드라이버의 등록이름에 대한 포인터이고, 다른 하나는 파일 연산 블럭에 대한 포인터이다. 이 파일 연산 블럭은, 파일을 열고, 쓰고, 읽고, 닫는 이런 파일 연산을 수행하는 문자 디바이스 드라이버에 있는 루틴의 주소들이다[85]. /proc/devices 에 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 파일이나 디렉토리처럼 각 장치 특수 파일은 VFS inode 로 표현된다. 문자 특수 파일에 대한 VFS inode 는 장치의 메이저 식별자와 마이너 식별자를 가지고 있다 (이는 모든 장치 특수 파일에서 동일하다). 이 VFS inode 는 장치 특수 파일을 조회한 경우에, 실제 기반하는 파일 시스템이 (EXT2 같은) 파일 시스템에 실제로 있는 정보를 가지고 만든다.

fs/ext2/inode.c ext2_read_inode () 참조

각 VFS inode 는 한 셋트의 파일 연산들과 연결되어 있는데, 이들 연산은 그 inode 가 가리키는 파일 시스템 객체에 따라 다르다[86]. 문자 특수 파일을 나타내는 VFS inode 가 만들어질 때마다, 이 inode 의 파일 연산 함수들은 기본 문자 장치 연산으로 설정된다. 이는 단 하나의 파일 연산 - 파일 열기 연산만 가지고 있다. 응용프로그램이 문자 특수 파일을 열면, 이 포괄적인 열기 파일 연산 함수는, 장치의 메이저 식별자를 chrdevs 벡터에 대한 인덱스로 사용하여, 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 매핑되어 불리게 된다.

def_chr_fops

fs/devices.c chrdev_open() 참조

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. blkdevs 는 chrdevs 벡터에서와 마찬가지로 장치의 메이저 장치번호로 인덱스되어 있다. 또한 그 엔트리 역시 device_struct 자료구조이다. 문자 장치와 다른 점은, 블럭 장치들의 클래스라는게 있다는 것이다. SCSI 장치나 IDE 장치 같은 것이 그런 클래스이다. 클래스는 리눅스 커널에 자신을 등록하고 커널에 파일 함수들을 제공한다. 어떤 클래스의 블럭 장치들에 사용하는 디바이스 드라이버는 클래스 고유의 특별한 클래스 인터페이스를 제공한다. 그래서 예를 들어 SCSI 디바이스 드라이버는, SCSI 서브시스템이 커널에 해당 장치에 대한 파일 함수를 제공하는데 사용할 수 있는 인터페이스를 SCSI 서브시스템에 제공해야 한다.

fs/devices.c 참조

모든 블럭 디바이스 드라이버는 보통의 파일 연산과 함께 버퍼 캐시에 대한 인터페이스를 제공해야 한다[87]. 각 블럭 디바이스 드라이버는 blk_dev 벡터에 있는 blk_dev_struct 자료구조의 내용을 채운다. 여기에서도, 이 벡터에 대한 인덱스는 장치의 메이저 번호이다.

drivers/block/ ll_rw_blk.c 참조

역주 85) 이 파일 연산 블럭을 나타내는 자료구조는 file_operations로, 여기에는 open, close, read, release 같은 기본적인 연산 외에도 lseek, ioctl, fsync 등의 여러 연산들이 더 있다. 이 자료구조는 블럭 장치의 디바이스 드라이버에도 사용된다. (flyduck)

역주 86) 이는 파일 시스템 객체에 따라서 다른 연산을 적용할 수 있게 하여, 파이프나 소켓같이 똑같이 파일 객체 인터페이스를 가지지만 실제로 다른 동작을 하는 것을 가능하게 한다. (flyduck)

역주 87) 디바이스 드라이버를 등록할 때 전달되는 file_operations 구조체에는 버퍼 캐시에 관련된 함수는 없다. 그래서 블럭 장치용으로 별도의 자료구조가 필요하게 되어, 버퍼 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 설정함으로써 이루어진다. (flyduck)

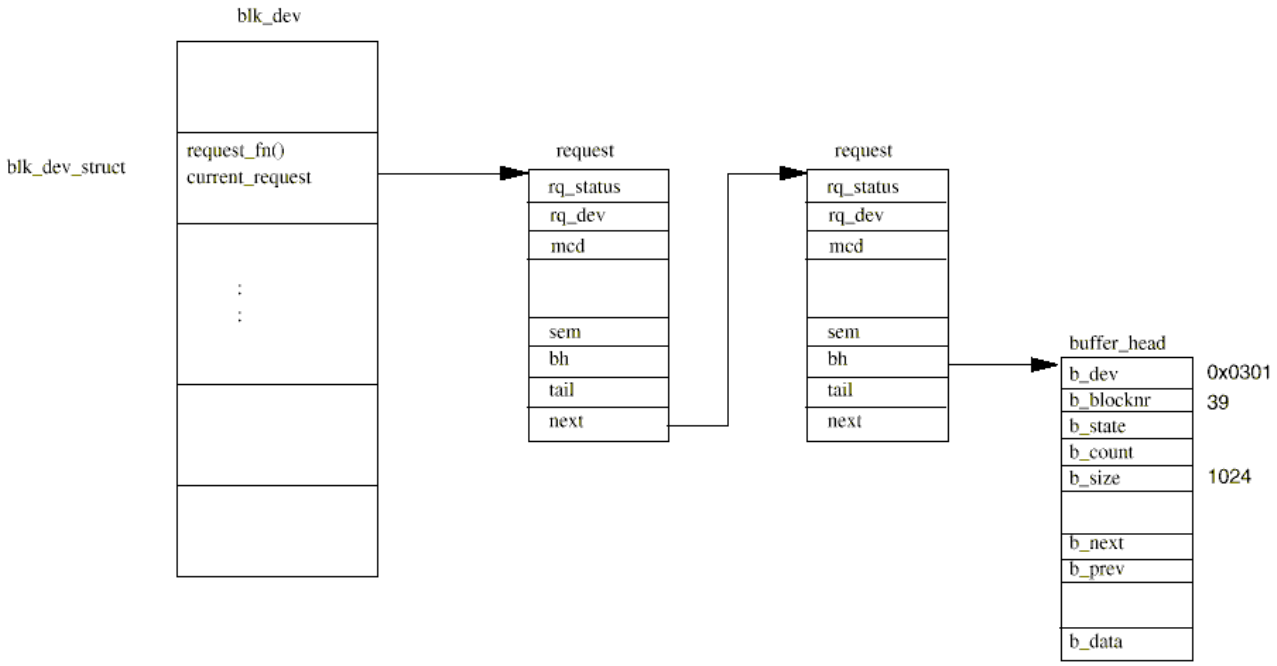


그림 8.2 : 버퍼 캐시 블럭 장치 요청

include/linux/blkdev.h 참조

`blk_dev_struct` 자료구조는 요청(request) 루틴의 주소와, 버퍼 캐시가 한 블럭의 데이터를 읽거나 쓰기 위해 드라이버에게 하는 요청을 나타내는 `request` 자료구조 리스트에 대한 포인터로 구성되어 있다.

버퍼 캐시는 등록된 장치에서 데이터를 읽거나 쓰려고 할 때, `request` 자료구조를 `blk_dev_struct` 에 추가한다. 그림 8.2 는 각 `request` 가 하나 이상의 `buffer_head` 자료구조에 대한 포인터를 가지고 있고, 각 `request` 는 한 블럭의 데이터를 읽거나 쓰라는 요청이라는 것을 보여준다. `buffer_head` 자료구조는 버퍼 캐시에 의해 락이 되며, 이 버퍼로의 블럭 연산이 끝나길 기다리는 프로세스가 있을 것이다. 각 `request` 구조체는 정적 리스트인 `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 `request` 를 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_head` 의 락을 해제하면, 그 블럭 연산이 끝나길 기다리며 잠들어있는 프로세스가 있을 때 이를 깨우게 된다. 이런 예로 파일 이름을 해결하는 과정에서 EXT2 파일 시스템이 파일 시스템을 담고 있는 블럭 장치로부터 다음 EXT2 디렉토리 엔트리를 포함하고 있는 데이터 블럭을 읽어야 하는 경우가 있다. 프로세스는 디바이스 드라이버가 자신을 깨울 때까지 잠들게 되며, 깨어났을 때에는 `buffer_head` 에 디렉토리 엔트리가 들어있을 것이다. 이제 `request` 자료구조는 비었다고 표시되고, 이 자료구조는 이제 다른 블럭 요청을 위해 사용될 수 있게 된다.

8.5 하드 디스크(Hard Disk)

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

디스크 드라이브는 하나 이상의 원반(platter)으로 구성되어 있고, 각 원반은 미세하게 간 유리나 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 10000RPM 까지 다르다. 이를 단지 360RPM 으로 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 물리적으로 원반의 표면을 건드리지 않으며, 대신 아주 얇은(백만분의 10 인치 정도) 공기 쿠션 위에 떠있다. 읽기/쓰기 헤드는. 헤드를 움직이는 장치(actuator)에 의해 원반의 표면을 가로질러 움직인다. 모든 읽기/쓰기 헤드는 서로 붙어 있어서 원반의 표면에서 똑같이 움직이게 된다.

원반의 각 표면은 트랙(track)이라고 하는 아주 가는 동심원으로 나누어진다. 트랙 0 은 가장 바깥에 있는 트랙이고, 가장 높은 번호를 갖는 트랙은 중심 축에 가장 가까운 트랙이다. 실린더(cylinder)는 똑같은 번호를 가지는 트랙의 집합이다. 따라서 디스크에 있는 모든 원반의 양쪽에 있는 5 번째 트랙은 모두 5 번 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 512 바이트이고, 디스크를 제작한 후 포맷을 할 때 이 크기가 지정된다.

디스크는 보통 기하적 구조 - 실린더와 헤드, 그리고 섹터의 개수 - 로 이야기한다. 예를 들어 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 1050 개의 실린더 (트랙), 16 개의 헤드 (8 개의 원반), 그리고 트랙마다 63 개의 섹터를 가지고 있다는 것을 말한다. 각 섹터 즉 블럭마다 512 바이트의 크기를 가지므로, 디스크의 저장용량은 529200 바이트가 된다. 이는 위에서 보여주는 디스크의 용량 516MB 하고는 차이가 있는데, 이는 섹터 중의 일부는 디스크 파티션 정보를 간직하는데 사용되기 때문이다. 어떤 디스크들은 자동으로 배드 섹터(bad sector)를 찾아내서 디스크가 제대로 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 쓰거나, 다른 목적으로 사용할 수 있도록 해준다. 많은 리눅스 시스템은 하나의 디스크에 세개의 파티션을 가진다 - 하나는 DOS 파일 시스템이고, 다른 하나는 EXT2 파일 시스템을, 마지막은 스왑 파티션이다. 하드디스크의 파티션 정보는 파티션 테이블에 적혀있다. 파티션 테이블의 각 엔트리는 파티션이 어디서 시작하고 어디서 끝나는지를 헤드와 섹터, 실린더 번호를 가지고 기술한다. fdisk 로 DOS 로 포맷된 디스크는 4 개의 1 차 디스크 파티션(primary disk partition)을 가질 수 있다. 파티션 테이블의 4 개 엔트리 모두가 쓰여야 하는 것은 아니다. fdisk 는 세가지 유형의 파티션을 지원하는데, 각각 1 차(primary), 확장(extended), 논리(logical) 파티션이다. 확장 파티션은 진짜 파티션이 아니라, 여러 개의 논리 파티션을 가지고 있는 것이다. 확장파티션과 논리 파티션은 1 차 파티션을 네개밖에 가질 수 있는 제한을 우회하기 위한 방법으로 개발되었다. 다음은 두개의 1 차 파티션을 가지고 있는 디스크에 대해 fdisk 를 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
```

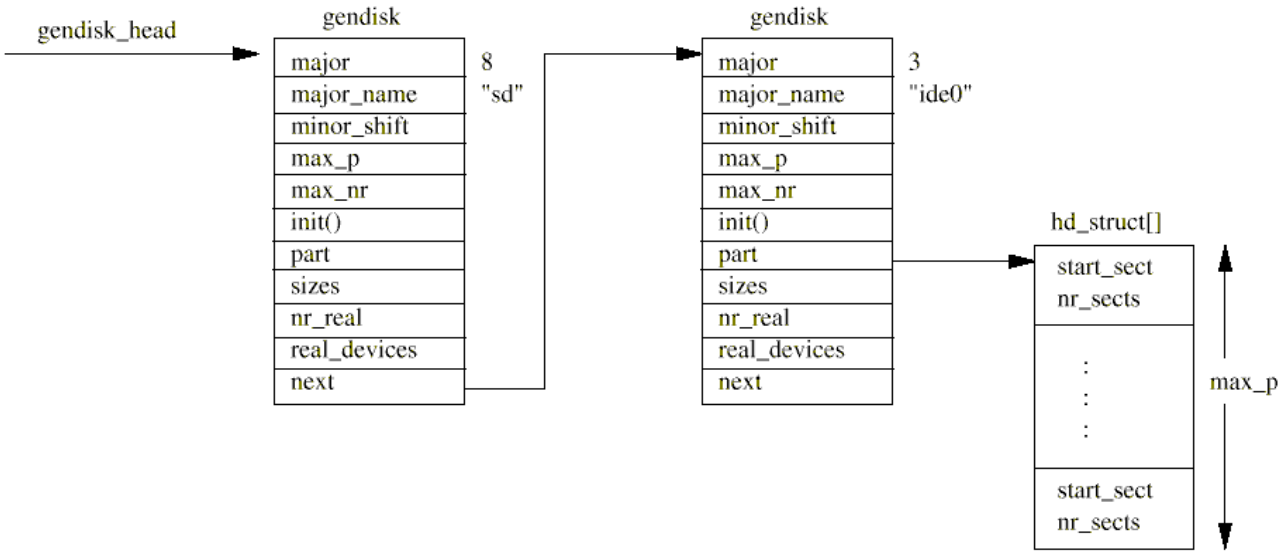


그림 8.3 : 디스크의 연결 리스트

```
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 1 에서 시작하며, 477 번 실린더, 32 개의 섹터, 63 개의 헤드까지 있다는 것을 보여준다. 여기엔 트랙당 32 개의 섹터와 64 개의 읽기/쓰기 헤드가 있으므로, 이 파티션의 크기는 있어서 실린더 개수와 같다[88]. fdisk 는 기본적으로 파티션을 실린더를 경계로 배열한다. 이는 맨 바깥 실린더 0 에서 시작하여 축이 있는 방향으로 안쪽으로 들어가 478 개의 실린더를 갖는다. 두번째 파티션은 스왑 파티션으로서 다음 실린더 (478)에서 시작하여 디스크의 가장 안쪽 실린더까지 뻗쳐있다.

리눅스는 초기화하는 동안 시스템에 있는 하드디스크의 배치도를 그려 이를 매핑한다. 먼저 시스템에 하드디스크가 몇 개가 있고 어떤 종류인지 알아낸다. 나아가 개별 디스크의 파티션이 어떻게 나누어졌는지도 찾아낸다. 이들은 `gendisk` 자료 구조로 표시되며, 이들의 리스트는 `gendisk_head` 리스트 포인터가 가리키고 있다. IDE 같은 개별 디스크 서브시스템은 초기화될 때 자신이 찾은 디스크를 `gendisk` 자료구조로 만들어낸다. 디스크 서브시스템은 이를 파일연산을 등록하고 엔트리를 `blk_dev` 자료구조에 넣을 때와 동시에 한다. 각 `gendisk` 자료구조는 고유한 메이저 장치 번호를 가지며, 이는 블럭 특수 장치의 메이저 번호와 일치한다. 예를 들어, SCSI 디스크 서브시스템은 모든 SCSI 디스크 장치에 적용되는 메이저번호 8 을 가지는 하나의 `gendisk` 엔트리 ("sd")를 만든다. 그림 8.3 은 두개의 `gendisk` 엔트리를 보여준다. 앞의 것은 SCSI 디스크 서브시스템의 것이고, 다음 것은 IDE 디스크 컨트롤러 것이다. 이것은 첫번째 IDE 컨트롤러인 `ide0` 이다.

디스크 서브시스템이 초기화할 때 만드는 `gendisk` 엔트리는, 단지 리눅스가 파티션을 검사할 때에만 쓰인다. 대신, 각 디스크 서브시스템은 장치의 메이저와 마이너 장치 번호를 물리적인 디스크에 있는 파티션과 매핑시킬 수 있도록 자신만의 자료구조를 구축한다. 블럭 장치가 버퍼 캐시나 파일 연산을 통해 읽혀지거나 쓰일 때, 커널은 이 연산을 블럭 장치 특수

역주 88) 한 실린더의 크기는 헤드의 수 * 섹터의 수 * 섹터 크기이므로 여기서는 64 * 32 * 512 = 1048576, 즉 1MB이다. (flyduck)

파일(예를 들어 /dev/sda2)에서 발견한 메이저 장치번호를 이용하여 올바른 장치로 보내게 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. IDE 는 SCSI 같은 I/O 버스가 아니라 디스크 인터페이스이다[89]. 각 IDE 컨트롤러는 두개까지 디스크를 지원할 수 있다. 하나는 주(master) 디스크이고 다른 하나는 종속(slave) 디스크이다. 주이냐 아니면 종속이냐는 보통 디스크에 있는 점퍼로 설정한다. 시스템에 있는 첫번째 IDE 컨트롤러는 1 차(primary) IDE 컨트롤러라고 하고 다음 것은 2 차(secondary) 컨트롤러라고 한다. IDE 는 1 초에 3.3 Mbyte 의 데이터를 전송할 수 있으며, IDE 디스크의 최대 크기는 538 MB 이다. 확장 IDE (Extended IDE, EIDE)는 디스크의 크기를 최대 8.6 GB, 전송속도를 초당 16.6 MB 까지 올린 것이다. IDE 와 EIDE 디스크는 SCSI 디스크보다 싸서 근래의 대부분의 PC 는 보드상에 하나 이상의 IDE 컨트롤러를 가지고 있다.

리눅스는 IDE 디스크의 이름을 디스크 컨트롤러를 발견한 순서에 따라 붙인다. 1 차 컨트롤러의 주 디스크는 /dev/hda, 종속 디스크는 /dev/hdb 이다. /dev/hdc 는 2 차 IDE 컨트롤러에 있는 주 디스크이다. IDE 서브시스템은 리눅스 커널에 IDE 컨트롤러를 등록하지만 디스크를 등록하는 것은 않는다. 1 차 IDE 컨트롤러에 대한 메이저 식별자(또는 장치 번호)는 3 이고, 2 차 IDE 컨트롤러는 22 이다. 그래서 시스템이 두개의 IDE 컨트롤러를 가지고 있다면, blk_dev 와 blkdevs 벡터의 3 번과 22 번 인덱스에 IDE 서브시스템의 엔트리가 있을 것이다. IDE 디스크의 블럭 특수 파일은 이런 번호 붙이는 방법을 반영하여, 1 차 IDE 컨트롤러에 연결되어 있는 /dev/hda 와 /dev/hdb 두 개는 메이저 식별자로 3 을 가진다. 커널은 이들 블럭 특수 파일을 관리하는 IDE 서브시스템에 대한 파일 연산이나 버퍼 캐시 연산을, 메이저 식별자를 인덱스로 사용하여 알아낸 IDE 서브시스템으로 전달한다. 어떤 요청이 들어왔을 때 어떤 IDE 디스크에게 요청이 들어왔는지 알아내는 것은 IDE 서브시스템의 몫이다. 이를 위해 IDE 서브시스템은 장치 특수 식별자에 있는 마이너 장치 번호를 사용하는데, 이것은 올바른 디스크의 해당하는 파티션으로 요청을 보낼 수 있도록 하는 정보를 가지고 있다. 1 차 IDE 컨트롤러에 있는 종속 IDE 드라이브인 /dev/hdb 의 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 PC 의 역사의 많은 부분을 함께 해왔다. 이 시간을 통해 이들 장치로의 인터페이스도 변해 왔으며, 이는 IDE 서브시스템의 초기화를 처음 생각했던 것보다 더 복잡하게 만든다.

리눅스가 지원할 수 있는 최대 IDE 컨트롤러의 갯수는 4 개이다. 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 하나는 종속 IDE 드라이브를 위한 것이다. IDE 서브시스템를 초기화할 때 리눅스는 먼저 시스템의 CMOS 메모리에 디스크 정보가 있는지 살펴본다. CMOS 메모리는 밧데리에서 전원을 공급받기 때문에 PC 의 전원을 끄더라도 내용물을 잃어버리지 않는 메모리다. 이 CMOS 메모리는, PC 의 전원이 켜져 있는지 꺼져 있는지에 무관하게 돌아가는 실시간 시계(real time clock, RTC) 장치에 들어있는 것이다. CMOS 메모리의 위치는 시스템의 BIOS 에서 설정하며, 이는 어떤 IDE 컨트롤러와 드라이브가 있는지 리눅스에게 알려준다. 리눅스는 발견한 디스크의 기하적 정보를 BIOS 로부터 얻으며, 이 정보를 이 드라이브에 대한 ide_hwif_t 자료구조를 설정하는데 사용한다. 최근에 나온 PC 들은 PCI EIDE 컨트롤러를 포함하고 있는 Intel

역주 89) SCSI는 여기에 디스크 외에도 스캐너같은 다른 외부장치를 붙일 수 있는 I/O 버스 규격이지만, IDE는 단지 디스크를 위한 인터페이스이다. (flyduck)

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_t` 가 설정된다. IDE 드라이버가 I/O 메모리 공간에 있는 IDE 명령 레지스터에 명령을 씀으로써 동작이 이루어진다. 1 차 IDE 컨트롤러의 명령 레지스터와 제어 레지스터의 기본 I/O 주소는 *0x1F0 - 0x1F7* 이다. 이들 주소는 IBM PC 초창기에서부터 관행으로 설정된 것이다. IDE 드라이버는 각 컨트롤러를 리눅스 블럭 버퍼 캐시와 VFS 에 등록하는데, 이는 `blk_dev` 와 `blkdevs` 벡터에 추가하는 것이다. IDE 드라이버는 또한 해당하는 인터럽트에 대한 제어권을 요청한다. 이들 인터럽트 역시 관행처럼 1 차 IDE 컨트롤러에 14, 2 차 IDE 컨트롤러는 15 로 설정된다. 그렇지만, 이들 설정은 IDE 의 다른 상세한 설정과 마찬가지로 커널에 명령행(command line) 옵션을 주어서 덮어 쓸 수 있다. IDE 드라이버는 또한 부팅시 발견된 IDE 컨트롤러마다 `gendisk` 를 만들어 `gendisk` 의 리스트에 추가한다 이 리스트는 나중에 부팅시 발견된 모든 하드 디스트의 파티션 테이블을 찾는데 사용한다. 파티션을 검사하는 코드는 IDE 컨트롤러가 두개의 IDE 디스트를 제어할 수도 있다는 것을 알고 있다.

8.5.3 SCSI 디스크

SCSI (Small Computer System Interface) 버스는, 하나 이상의 호스트를 포함하여 버스마다 8 개까지의 장치를 지원할 수 있는 효율적인 1 대 1 데이터 버스이다. 각 장치는 고유한 식별자를 가져야 하는데, 이는 대개 디스크에 있는 점퍼로 설정한다. 버스에 있는 어떤 두 장치 사이이든간에 동기적으로 또는 비동기적으로 데이터를 전송할 수 있으며, 32 비트 크기로 초당 40 MB 까지 전송할 수 있다. SCSI 버스는 장치간에 데이터와 상태 정보를 함께 전송하며, 전송 시작자(initiator)와 전송 대상(target) 사이의 하나의 트랜잭션은 여덟개의 서로 다른 상태를 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 받을 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 시작자와 전송 대상 사이에 전달된다.

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

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

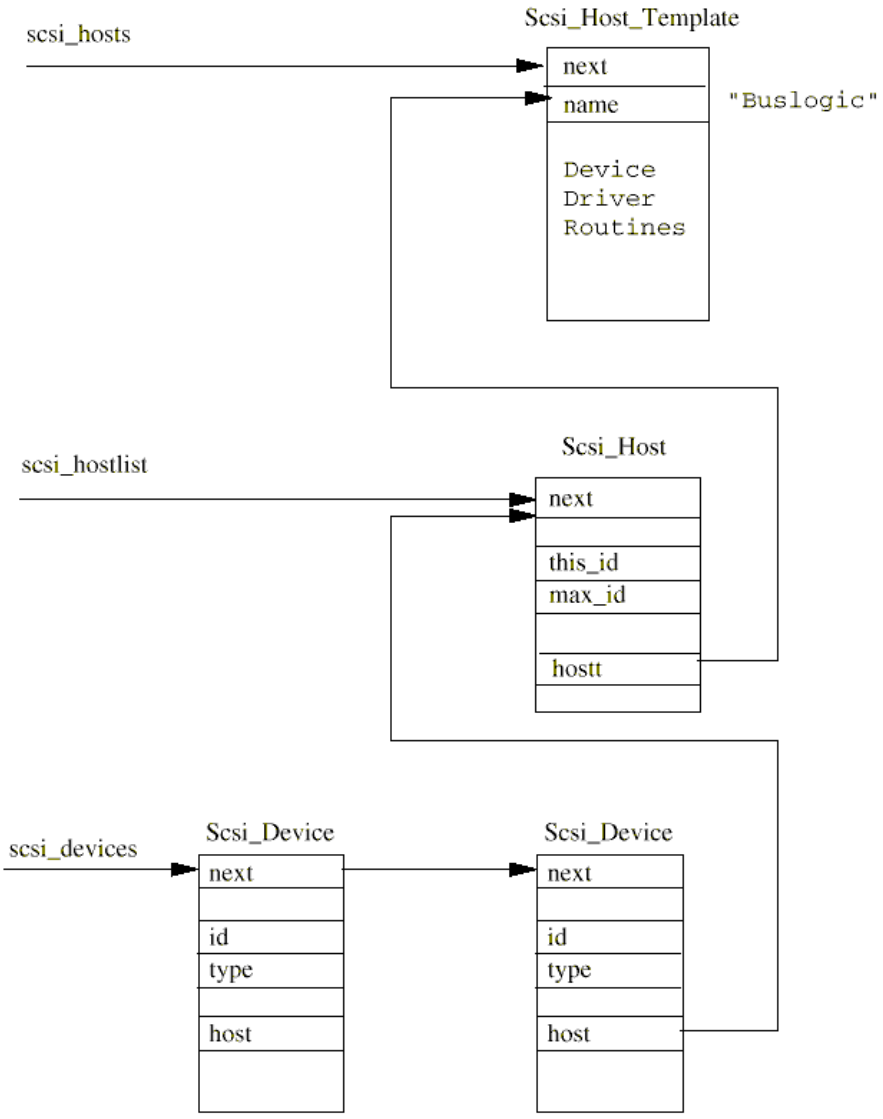


그림 8.4 : SCSI 자료구조

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 컨트롤러를 하나 이상 가지고 있다면, 각각은 별도의 SCSI 호스트로 표현된다. 이말은 SCSI 디바이스 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-ROM 이나 테이프같은 제거가능한 매체를 가진 경우 리눅스는 그 매체가 제거되었는지 인식할 수 있어야 한다. 다른 디스크 종류는 다른 메이저 장치 번호를 가지며, 이는 리눅스가 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 복잡하다. 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 장치 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 감지하는 것 같은 일을 수행하는 SCSI 호스트 별로 고유한 루틴들에 대한 포인터를 가지고 있다. SCSI 서브시스템은 자신을 설정하는 동안 이들 루틴을 부르며, 이들 루틴은 이런 호스트 유형을 지원하는 SCSI 디바이스 드라이버의 한 부분이다. 실제 SCSI 장치가 연결되어 있는 감지된 SCSI 호스트는, 자신의 `Scsi_Host_Template` 자료구조를 활성화된 SCSI 호스트들의 `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 하나씩해서, 리스트에 두개의 `Scsi_Host` 엔트리를 가지게 된다. 각 `Scsi_Host` 는 자신의 디바이스 드라이버를 나타내는 `Scsi_Host_Template` 를 가리킨다.

이제 모든 SCSI 호스트를 발견했으므로, SCSI 서브시스템은 어떤 SCSI 장치가 호스트의 버스에 연결되었는지 찾아야 한다. SCSI 장치는 0 에서 7 까지의 장치 번호를 가지는데, 이 번호는 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 리눅스에게 제작자(vendor)의 이름과 장치의 모델명과 개정(revision) 이름을 전달한다. SCSI 명령은 `Scsi_Cmnd` 자료구조로 표현되고, 디바이스 드라이버의 `Scsi_Host_Template` 자료구조에 있는 디바이스 드라이버 루틴에 부를 대 이를 전달한다. 발견한 모든 SCSI 장치는 `Scsi_Device` 자료구조로 표현하며, 각각은 자신의 부모 `Scsi_Host` 를 가리킨다. 모든 `Scsi_Device` 자료구조는 `scsi_devices` 리스트에 추가된다. 그림 8.4 는 어떻게 이들 주된 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 주어진 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 블럭 연산(파일이나 버퍼캐시)을 올바른 디바이스 드라이버나 SCSI 호스트로 보내는데 사용한다. 각 SCSI 유형은 `Scsi_Device_Template` 자료구조로 표현한다. 이 자료구조는 이 유형의 SCSI 장치에 대한 정보와 다양한 작업을 수행하는 루틴들의 주소를 가지고 있다. SCSI 서브시스템은 이들 템플릿을 사용하여 각 유형의 SCSI 장치에 대해 SCSI 유형에 따른 루틴을 부르는데 사용한다. 다르게 말하면, SCSI 서브시스템이 SCSI 디스크 장치를 연결하려고 할 때, SCSI 디스크 유형에 따른 연결 루틴을 부른다는 것이다. 어떤 유형을 갖는 SCSI 장치가 하나 이상 발견되면 `Scsi_Type_Template` 자료구조가 `scsi_Devicelist` 리스트에 추가된다.

SCSI 서브시스템 초기화의 마지막 상태는 등록된 각 `Scsi_Device_Template` 의 종료(finish) 함수를 부르는 것이다. SCSI 디스크 유형이라면, 이는 발견한 모든 SCSI 디스크를 회전시켜 그들의 디스크 구조를 기록하는 일을 한다. 또한 그림 8.3 에 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. 정상적으로 동작하는 각 SCSI 장치 유형은 자신을 커널에 등록하여, 리눅스가 블럭 장치 요청을 자신에게 보낼 수 있게 한다. 여기에는 `blk_dev`를 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 있으며, 이 벡터는 SCSI 디스크 파티션의 마이너 장치 번호 중 일부를 사용하여 인덱스가 되어 있다. 예를 들어 `/dev/sdb1`은 8 번의 메이저 번호와 17 번의 마이너 번호를 가지며, 이는 인덱스 1 을 생성한다. 각 `Scsi_Disk` 자료구조는 이 장치를 나타내는 `Scsi_Device` 자료구조에 대한 포인터를 갖고 있다. `Scsi_Device` 자료구조는 차례로 자신을 "소유"하고 있는 `Scsi_Host` 자료구조를 가리키고 있다. 버퍼 캐시로부터 온 `request` 자료구조는 SCSI 장치로 보내야 하는 SCSI 명령을 기술하는 `Scsi_Cmnd`[90] 구조체로 바뀌고, 이는 이 장치를 나타내는 `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 생각한다. 이는 대개의 경우 이더넷 카드같은 물리적인 장치이다. 어떤 네트웍 장치는 소프트웨어로만 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[91]. 이 `device` 자료구조는 장치에 대한 정보와, 리눅스에서 지원하는 다양한 종류의 네트웍 프로토콜이 장치의 서비스를 이용할 수 있도록 하는 함수들의 주소를 가지고 있다. 이 함수들은 대부분 네트웍 장치를 통한 데이터 전송과 관계가 있다. 장치는 표준 네트웍 지원 매커니즘을 사용하여 전송받은 데이터를 올바른 프로로콜 계층으로 전달한다. 보내고 받는 모든 네트웍 데이터(패킷)은 `sk_buff` 자료구조로 표현되는데, 이는 네트웍 프로토콜 헤더를 쉽게 첨가하거나 제거할 수 있도록 만들어진 유연한 자료구조이다. 네트웍 프로토콜 계층이 어떻게 네트웍 장치를 사용하는지, 어떻게 `sk_buff` 자료구조를 가지고 데이터를 앞뒤로 전달하는 지는 네트웍 장(10 장)에서 상세하게 다룬다. 여기서는 `device` 자료 구조와 네트웍 장치를 어떻게 발견하고 초기화하는지에 중심을 둔다.

include/linux/netdevice.h 참조

`device` 자료구조는 다음과 같은 네트웍 장치에 대한 정보를 가진다.

이름 특수 장치 파일이 `mknod` 명령으로 만들어지는 블럭 장치나 문자 장치와는 달리, 네트웍 장치 특수 파일은 시스템에 있는 네트웍 장치를 발견하고 초기화하는 과정에서 차례로 나타난다. 이들의 이름은 장치의 유형을 나타내는 그런 표준적인 이름이다. 같은 유형의 장치들에는, 0 부터 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

역주 90) 원문에는 `Scsi_Cmd`로 되어 있지만 `Scsi_Cmnd`가 맞다. (flyduck)

역주 91) 앞에서 설명한바와 같이 실제로 제어할 장치가 있을 때 이를 등록한다는 것이다. (flyduck)

/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 레지스터와 상태 레지스터가 있는 I/O 메모리 상의 주소이다. DMA 채널(DMA channel)은 네트웍 장치가 사용하는 DMA 채널 번호이다. 이 모든 정보는 부팅시에 장치를 초기화할 때 설정된다.

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

IFF_UP	인터페이스가 위에 있고(up) 실행중이다.
IFF_BROADCAST	`device` 의 브로드캐스트 주소가 유효하다.
IFF_DEBUG	장치 디버깅 옵션이 켜져 있다.
IFF_LOOPBAK	루프백 장치이다.
IFF_POINTTOPOINT	SLIP 이나 PPP 같은 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)[92] 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 IP 같은 프로토콜 계층이 전송에 사용할 적당한 패킷의 크기를 선택하기 위해 사용한다.

계열(Family) 이것은 장치가 지원할 수 있는 프로토콜 계열을 나타낸다. 모든 리눅스 네트웍 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 계층에서 호출할 수 있는 표준 함수 집합을 제공한다. 이는 셋업하고 프레임을 전송하는 루틴뿐만 아니라, 표준 프레임 헤더를 추가하고, 통계정보를 모으는 루틴도 포함한다. 이 통계정보는 ifconfig 명령으로 볼수 있다.

8.6.1 네트웍 장치 초기화

네트웍 디바이스 드라이버는 다른 리눅스 디바이스 드라이버와 마찬가지로 커널에 직접 포

역주 92) IP는 기본적으로 시작주소 하나와 목적지 주소 하나를 가지고 있다. 즉 어떤 곳에서 단 하나의 목적지로만 IP 패킷을 보낼 수 있다는 것이다. 이는 화상회의같이 같은 패킷을 여러 목적지로 보내는 경우 중복된 내용을 수신하는 숫자만큼 목적지를 따로 지정해 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 목적지로 패킷을 전송하게 해 주는 프로토콜이다. (flyduck)

함되어 있을 수 있다. 각 잠재적인[93] 네트웍 장치는 dev_base 리스트 포인터가 가리키는 네트웍 장치 리스트에 있는 device 자료구조로 표현된다. 네트웍 계층은 장치에 고유한 작업을 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 커널에 포함된 모든 네트웍 디바이스 드라이버가 자신이 제어할 장치를 갖는 것은 아니라는 것이다. 그리고 두번째로 시스템에 있는 이더넷 장치는 밑에 있는 디바이스 드라이버가 어떤 거든지간에 항상 /dev/eth0, /dev/eth1 과 같이 나타난다는 것이다. 먼저 "없는" 네트웍 장치 문제는 쉽게 풀 수 있다. 각 네트웍 장치의 초기화 루틴을 부르면, 이 루틴은 자신이 구동할 컨트롤러를 찾았는지 못찾았는지 의미하는 상태값을 돌려준다. 만약 드라이버가 아무런 장치도 찾지 못했다면, dev_base 가 가리키고 있는 device 리스트에 있는 엔트리가 제거된다. 만약 드라이버가 장치를 찾게 된다면, 드라이버는 device 자료구조의 나머지 부분을 장치에 대한 정보와 네트웍 디바이스 드라이버에 있는 드라이버가 지원하는 함수들의 주소로 채운다.

두번째 문제는 이더넷 장치를 표준 /dev/ethN 장치 특수파일에 동적으로 부여하는 문제로 이는 좀더 우아한 방법으로 해결된다. 장치 목록에는 eth0 부터 eth7 까지 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 자신이 제어할 물리적인 하드웨어를 초기화하고, 어떤 IRQ 를 사용하고 있고 어떤 DMA 채널을 사용하고 있는지 (만약 있다면) 등등을 알아낸다. 드라이버는 자신이 제어할 네트웍 장치를 여러 개를 찾을 수 있는데, 이 경우 드라이버는 여러 개의 /dev/ethN device 자료구조을 넘겨준다. 여덟개의 표준 /dev/ethN 이 모두 할당되면, 더 이상 이더넷 장치를 찾지 않는다.

번역 : 이호, 신문석
정리 : 이호

역주 93) 장치가 있을 가능성은 있지만 아직 확인한 것은 아니기에 잠재적으로 존재한다. (flyduck)

9장

파일 시스템 (File System)

이 장은 리눅스 커널이 지원하는 파일 시스템 안의 파일들을 어떻게 관리하는가를 설명한다. 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 함으로써 리눅스는 유연성을 갖게 되었고 다른 많은 운영체제와 잘 공존할 수 있게 되었다. 리눅스를 처음 만들었을 때는 `ext`, `ext2`, `xia`, `minix`, `umsdos`, `msdos`, `vfat`, `proc`, `smb`, `ncp`, `iso9660`, `sysv`, `hpfs`, `affs`, `ufs`의 15가지 파일 시스템을 지원했고, 당연히 시간이 지남에 따라 더 많은 것이 추가되었다.

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

디스크가 초기화될 때 (가령, `fdisk`를 사용하여) 디스크는 파티션 구조를 가지게 되는데, 이것은 물리적인 디스크를 논리적으로 몇 개의 파티션으로 분리하는 것이다. 각 파티션은 하나의 파일 시스템 - 예를 들어, EXT2 파일 시스템 - 을 가지게 된다. 파일 시스템은 물리적인 장치의 블럭에, 파일을 디렉토리나 소프트 링크 등과 함께 논리적인 계층 구조로 구성한다. 파일 시스템을 담을 수 있는 장치는 블럭 장치이다. 시스템에 있는 첫번째 IDE 디스크의 첫번째 파티션인 `/dev/hda1`은 블럭 장치이다. 리눅스 파일 시스템은 이러한 블럭 장치들을 단순히 일렬로 늘어놓은 블럭의 모음으로 간주하며, 그 밑에 있는 물리적인 디스크에 대해서는 알지도 못하고 상관도 하지 않는다. 장치의 특정 블럭을 읽으라는 요구를 그 장치에게 의미있는 요소들, 즉 특정 트랙, 섹터, 실린더 등 하드 디스크상에 그 블럭이 있는 위치로 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 조작되는 다른 물리적 매체에 있는 것은 전혀 문제가 되지 않는다 (적어도 시스템 사용자에게는 그렇다). 파일 시스템은 심지어 로컬 시스템에 있지 않을 수도 있다. 네트웍 연결로 원격지에서 마운트된 디스크일 수도 있는 것이다. 리눅스 시스템이 SCSI 디스크에 루트 디렉토리를 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 디스크이다. 첫번째 IDE 컨트롤러는 PCI이며, 두번째 것은 IDE CDROM도 제어하는 ISA 컨트롤러라는 것은 전혀 상관이 없다. 나는 모뎀과 PPP 네트웍 프로토콜을 사용해서 내가 일하는 곳에 전화를 걸어, 내 알파 AXP 리눅스 시스템의 파일 시스템을 /mnt/remote에 원격으로 마운트할 수도 있다.

파일 시스템의 파일들은 데이터의 집합이다. 이 장은 filesystems.tex라는 아스키 파일에 들어 있다. 파일 시스템은 파일에 담긴 데이터 뿐만 아니라, 파일 시스템의 구조도 가지고 있다. 파일 시스템의 구조에는 리눅스의 사용자나 프로세스가 볼 수 있는 파일, 디렉토리에 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[94].

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

리눅스에 EXT 파일 시스템이 추가되었을 때, 중요한 발전이 있었다. 실제 파일 시스템이 가상 파일 시스템(Virtual File System, VFS)이라는 인터페이스 계층을 통해서 운영체제와 운영체제의 서비스로부터 분리된 것이다. VFS 덕분에 리눅스는 VFS를 지원하는 서로 다른 많은 파일 시스템들을 사용할 수 있게 되었다. 리눅스 파일 시스템의 모든 세세한 것들이 소프트웨어에 의해서 변환되어서, 모든 파일 시스템이 리눅스 커널의 다른 부분들과 그 위에서 실행되는 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 구현되었다. 또한, 파일과 그 안의 자료가 정확하게 유지될 수 있어야만 한다. 이러한 두가지 요구 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 실행중인 커널안에서 파일 시스템의 자료구조들을 볼 수 있다면, 파일 시스템이 자료들을 읽거나 쓰는 것도 볼 수 있을 것이다. 접근하려는 파일이나 디렉토리를 나타내는 자료구조는 디바이스 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 버퍼를 캐시할 뿐만 아니라, 블럭 디바이스 드라이버와의 비동기적인 인터페이스의 관리도 도와준다.

9.1 2차 확장 파일 시스템 (EXT2)

2차 확장 파일 시스템은 리눅스를 위한 확장성있고 강력한 파일 시스템으로 Rémy Card가

94) 음, 고의는 아니었겠지만, 나는 리눅스가 가진 개발자보다 많은 변호사를 가진 운영체제에 물려왔다.

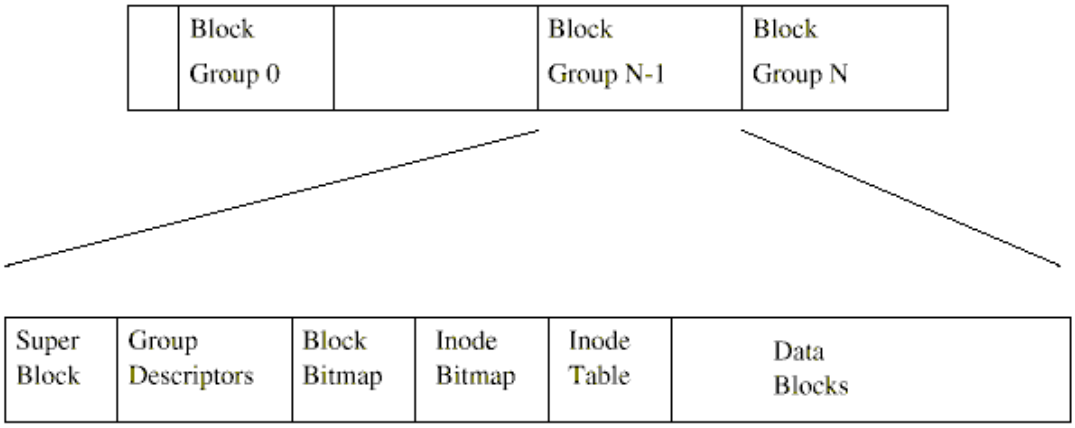


그림 9.1 : EXT2 파일 시스템의 물리적 배치도

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 크기는 (mke2fs 명령을 통해) 파일 시스템이 만들어 질 때 결정된다. 모든 파일의 크기는 블럭의 크기에 따라 올림이 된다. 만약 블럭의 크기가 1024바이트일 때, 크기가 1025바이트인 파일은 1024바이트 블럭 두개를 차지하게 된다. 이는 파일 하나당 평균 블럭의 절반 크기만큼을 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 구조를 표현하는 정보를 담고 있어야 한다. EXT2는 파일 시스템 배치도를 정의하기 위하여 시스템내의 각 파일을 inode 자료구조로 표현한다. inode는 파일의 데이터가 어느 블럭에 들어있는지, 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 디렉토리에 속하는 파일들의 inode에 대한 포인터를 갖고 있다.

fs/ext2/* 참조

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 시스템을 담고 있는 블럭 장치로부터 정보나 데이터를 읽으려고 한다면 단지 해당 디바이스 드라이버에게 몇 개의 블럭을 읽어달라고 요청하기만 하면 된다. EXT2 파일 시스템은 자신이 차지하고 있는 논리적인 파티션을 다시 블럭 그룹으로 쪼갠다. 각 블럭 그룹은 파일 시스템에서 무결성의 핵심이 되는 정보를 중복해서 갖고 있으며, 실제 파일과 디렉토리를 정보와 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.

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

include/linux/ ext2_fs_i.h 참조

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 포인터들의 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 실제 파일은 아니지만 장치를 액세스하는데 사용되는 프로그램을 다룬다. `/dev` 디렉토리 아래의 모든 장치 파일은 프로그램이 리눅스 장치를 액세스할 수 있도록 하기 위하여 거기에 있는 것이다. 예를 들어 마운트 프로그램은 마운트하려는 장치 파일을 인자로 사용한다.

9.1.2 EXT2 수퍼블럭(Superblock)

include/linux/ ext2_fs_sb.h 참조

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

매직 넘버(Magic Number) 이 값은 마운트하는 소프트웨어로 하여금 이것이 진짜 EXT2 파일 시스템의 수퍼블럭이라는 것을 확인케한다. 현재 버전의 EXT2에서는 0xEF53으로 되어 있다.

개정 레벨(Revision Level) 이 값은 메이저 개정 레벨과 마이너 개정 레벨로 구성되며, 마운트 프로그램이 어떤 특정한 버전에서만 지원되는 기능이 이 파일 시스템에서 지원되는지 아닌지를 확인하는데 사용된다. 또한 기능 호환성 항목라는 것이 있어서 마운트 프로그램이 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 횟수와 같아지면 "최대 마운트 횟수에 도달하였습니다, e2fsck를 실행하는 것이 좋습니다[95]" 라는 메시지가 표시된다.

블럭 그룹 번호(Block Group Number) 현재 보고 있는 수퍼블럭 복제본을 갖고 있는 블럭 그룹의 번호.

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

블럭 크기(Block Size) 이 파일 시스템의 블럭 크기를 바이트 단위로 (예를 들어, 1024 바이트) 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 블럭 그룹을 위한 그룹 기술자는 각 블럭 그룹에 복제되어 파일 시스템이 파괴되는 경우를 대비한다. 각 그룹 기술자는 다음과 같은 정보를 갖고 있다 :

include/linux/ext2_fs.h 에서 ext2_group_desc 참조

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 시스템에서 실제로 사용되는 것은 (블럭 그룹 0에 있는) 첫번째 복사본 뿐이다. 다른 복사본들은, 수퍼블럭의 복사본들과 마찬가지로, 원본이 깨질 경우를 대비하고 있다.

9.1.4 EXT2 디렉토리

EXT2 파일 시스템에서 디렉토리는 파일 시스템내의 파일에 대한 접근 경로를 만들고 저장하는 특별한 파일이다. 그림 9.3은 메모리 상에서의 디렉토리 엔트리의 모양을 보여준다. 디렉토리 파일은 디렉토리 엔트리의 리스트이며 각각의 디렉토리 엔트리는 다음과 같은 정보를 갖고 있다 :

include/linux/ext2_fs.h 에서 ext2_dir_entry 참조

inode 이 디렉토리 엔트리에 해당하는 inode. 이 값은 블럭 그룹의 inode 테이블에 저장되어 있는 inode 배열에 대한 인덱스이다. 그림 9.3 에서 `file`이라는 이름의 파일에 대한 디렉토리 엔트리는 `i1`이라는 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 "

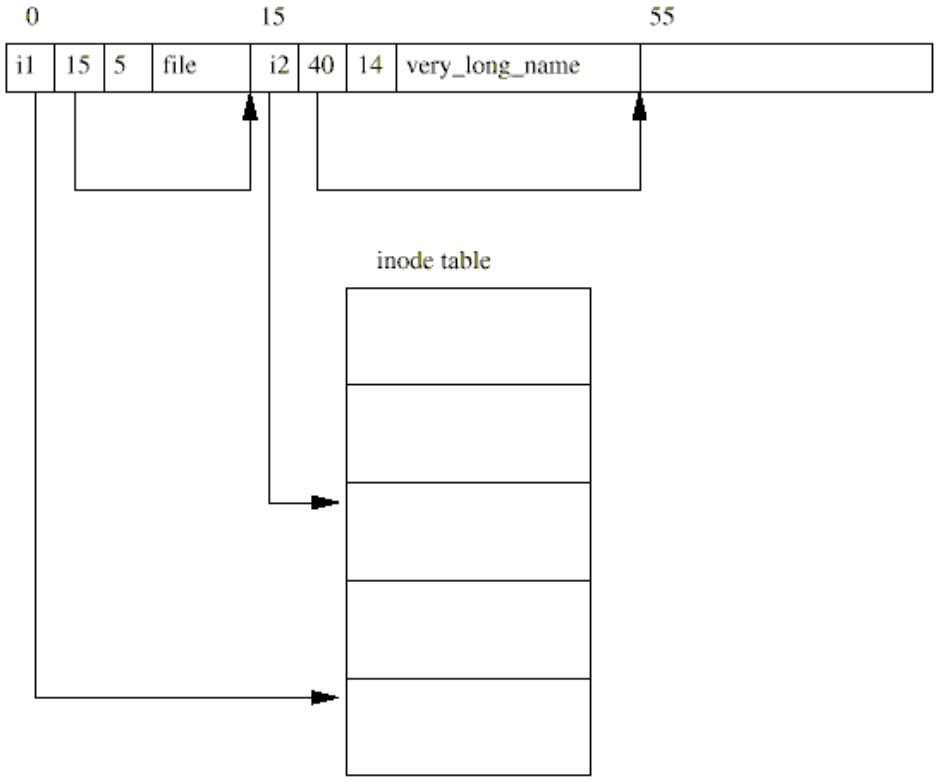


그림 9.3 : EXT2 디렉토리

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 슬래시('/')가 붙은 디렉토리 이름이 이어지고 마지막에 파일 이름이 오는 형태이다. 예를 들어, 파일 이름이 `/home/rusling/.cshrc`이라면, `/home`과 `/rusling`은 디렉토리 이름이고 파일 이름은 `.cshrc`이다. 다른 모든 유닉스 시스템과 마찬가지로 리눅스는 파일 이름 자체의 형식은 신경쓰지 않는다. 길이 제한도 없고, 인쇄 가능한 아무런 문자로 구성된다. 이 파일을 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 찾아야 한다. 예를 들어 루트 inode의 번호가 42라면 우리는 블럭 그룹 0의 inode 테이블의 42번째 inode가 필요한 것이다. 루트 inode는 EXT2 디렉토리를 위한 것이다. 다시 말해서 루트 inode의 모드는 루트 inode가 디렉토리임을 나타내며 데이터 블럭에는 EXT2 디렉토리 엔트리가 들어 있다.

`home`은 여러 디렉토리 엔트리 중의 하나일 뿐 이며 `/home` 디렉토리를 나타내는 inode의 번호를 알려준다. 이 디렉토리를 읽어서 (디렉토리를 읽으려면 우선 inode를 읽고 그 inode가 가리키는 데이터 블럭으로부터 디렉토리 엔트리를 읽어야 한다.) `rusling` 엔트리를 찾으면 그 엔트리는 `/home/rusling` 디렉토리를 나타내는 inode의 번호를 알려줄 것이다. 마침내 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 떨어질수록 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 할당하려고 한다. 둘 다 실패했을 때만 다른 블럭 그룹에 있는 데이터 블럭을 할당한다.

프로세스가 파일에 데이터를 쓰려고 할 때마다, 리눅스 파일 시스템은 데이터가 파일에 마지막으로 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 것이다. 할당과 해제는 수퍼블럭에 있는 항목을 변경하며, 리눅스 파일 시스템은 둘 이상의 프로세스가 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. 수퍼블럭의 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 블럭이 충분히 남아있는지 확인한다. 만약 프리 블럭이 충분하지 않다면 더 이상 할당받으려는 시도는 실패할 것이기 때문에 프로세스는 이 파일 시스템의 수퍼블럭에 대한 통제권을 내놓게 된다.

fs/ext2/balloc.c ext2_new_block () 참조

만약 파일 시스템에 프리 블럭이 충분하면 프로세스는 할당을 받게 된다. 만약 EXT2 파일 시스템이 데이터 블럭을 미리 할당하도록 만들어졌다면 미리 할당된 블럭을 사용할 수도 있다. 미리 할당된 블럭은 실제로 존재하지는 않지만 할당된 블럭 비트맵에 예약되어 있다. 우리가 새로운 데이터 블럭을 할당해 주려고 하는 파일을 나타내는 VFS inode는 EXT2 고유의 항목 두개를 갖고 있다. `prealloc_block`은 처음에 미리 할당된 데이터 블럭의 수를 나타내고, `prealloc_count`는 그 중에서 몇 개가 남아 있는지를 나타낸다. 미리 할당된 블럭이 없거나 블럭을 미리 할당하는 기능이 사용되지 않고 있으면, EXT2 파일 시스템은 새로운 블럭을 할당하여야만 한다. EXT2 파일 시스템은 우선 파일의 마지막 데이터 블럭의 다음 데이터 블럭이 비었는지 본다. 논리적으로 보아 이 블럭은 순차식 액세스를 더욱 빠르게 해주기 때문에 가장 효율적인 블럭이다. 만약, 그 블럭이 비어있지 않다면 검색의 범위를 넓혀서 가장 이상적인 블럭에서 64블럭 이내의 데이터 블럭을 살펴본다. 이러한 블럭은 비록 가장 이상적이지는 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 그룹을 뒤지게 된다. 블럭 할당 프로그램은 한 블럭 그룹 안에서 여덟개의 빈 데이터 블럭으로 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 캐시 내에 데이터 버퍼를 할당한다. 그러한 데이터 버퍼는 파일 시스템을 지원하는 장치 식별자와 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 있는 프로세스 중 첫번째 프로세스가 다시 실행되게 되며 파일 처리에 필요한 수퍼블럭의 독

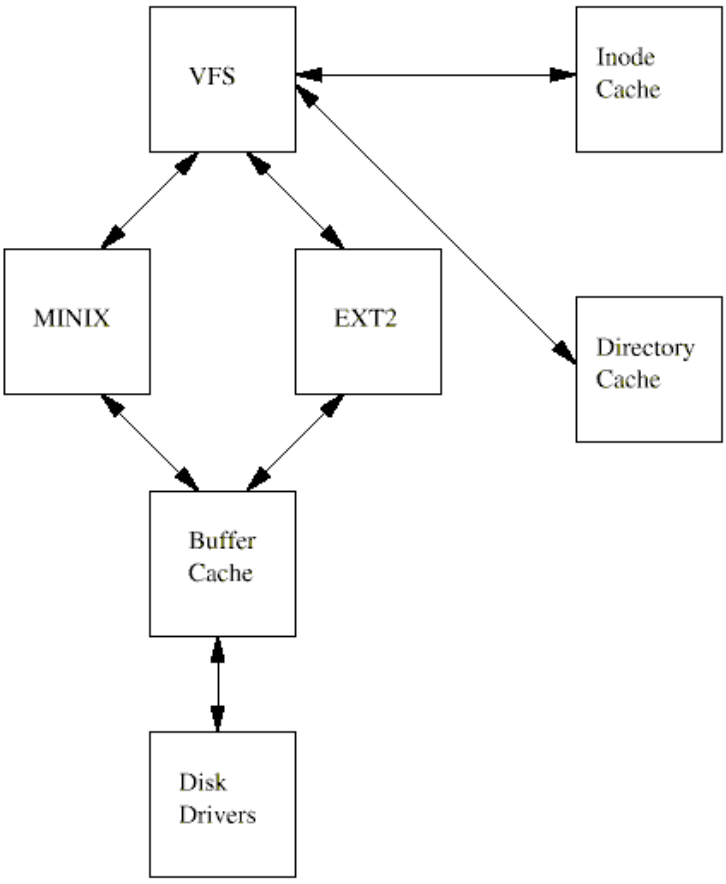


그림 9.4 : 가상 파일 시스템의 논리적 구성도

점적 통제를 갖게 된다. 프로세스의 데이터는 데이터 블럭이 다 채워지면 또 새로운 데이터 블럭에 기록되며 이러한 과정은 데이터 블럭이 할당될 때마다 똑같이 반복된다.

9.2 가상 파일 시스템(Virtual File System, VFS)

fs/* 참조

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. 파일 시스템 모듈은 시스템이 필요로 할 때 로드된다. 예를 들어, VFAT 파일시스템이 커널 모듈로 되어 있다면, VFAT 파일 시스템이 마운트될 때 로드될 것이다. 블럭장치에 기반한 파일 시스템이 마운트되고, 이것이 루트 파일 시스템을 포함하고 있다면, VFS는 이것의 수퍼블럭을 읽어야 한다. 파일 시스템 타입별 수퍼블럭 읽기 루틴은 파일 시스템의 배치도를 정확히 알 수 있어야 하며, 그 정보를 VFS 수퍼블럭 자료구조에 매핑 시킬 수 있어야 한다. VFS는 마운트된 파일 시스템과 VFS 수퍼블럭의 리스트를 관리한다. 각각의 VFS 수퍼블럭은 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 루틴과 마찬가지로 VFS inode의 각 항목을 채운다. 각각의 VFS 수퍼블럭은 파일 시스템의 첫번째 VFS inode에 대한 포인터를 갖고 있다. 루트 파일 시스템의 경우 이 inode는 "/" 디렉토리를 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 탐색하는 시스템 루틴을 부르게 된다. 예를 들어, 어떤 디렉토리에 대해 ls 명령을 치거나 어떤 파일에 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 해당 inode를 읽어들이기 위하여 각 파일 시스템 고유의 루틴을 호출하여야 한다. 이렇게 읽어들인 inode는 inode 캐시에 저장되어 다음번 액세스할 때에는 캐시에서 찾을 수 있게 된다. 덜 사용되는 VFS inode는 캐시로부터 제거된다.

fs/inode.c 참조

모든 리눅스 파일 시스템은 파일 시스템을 갖고 있는 실제 장치에 대한 액세스 속도를 높이기 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, 블럭 기반의, 일반적으로는 비동기적인 인터페이스를 제공한다. 심지어 SCSI 장치와 같이 비교적 복잡한 블럭장치도 이렇게 한다. 실제 파일 시스템이 그 아래에 깔려있는 실제 디스크에서 데이터를 읽게 되면 이는 블럭 디바이스 드라이버로 하여금 자신이 컨트롤하는 장치로부터 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 필요하게 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.

fs/buffer.c 참조

VFS에서는 자주 사용되는 디렉토리의 inode를 빨리 찾기 위하여 디렉토리 찾아보기 캐시도 갖고 있다. 실험삼아 최근에 리스트를 본 적이 없는 디렉토리의 리스트를 보려고 해보라. 처음에 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.

fs/dcache.c 참조

9.2.1 VFS 수퍼블럭

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

include/linux/fs.h 참조

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[96] - 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` 자료구조를 가리키는 포인터이다.

파일 시스템 고유 정보(File System Specific) 이 파일 시스템이 필요로 하는 정보를 가리키는 포인터.

9.2.2 VFS inode

include/linux/fs.h 참조

EXT2 파일 시스템과 마찬가지로, VFS 안에 있는 모든 파일, 디렉토리 등은 반드시 단지 하나의 VFS inode로 표현된다[97]. 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 눈여겨 볼 만한 정보는 다음과 같다.

장치(Device) 이것은 이 VFS inode가 나타내는 파일 또는 다른 어떤 것을 가지고 있는 장치의 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 나타내는가와 접근 권한 등을 나타낸다[98].

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 나타낸다.

inode 연산(inode operations) 연산 루틴의 주소들에 대한 포인터이다. 이들 루틴은 파일 시스템마다 고유하며 inode에 대한 여러가지 연산, 예를 들어 이 inode가 나타내는 파일의 제거과 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)

역주 96) 리눅스를 기본으로 설치하면 블럭 크기는 512바이트이다. (심마로)
역주 97) 1대1 관계이다. (심마로)
역주 98) chmod가 변경하는 항목이 이것이다. (심마로)

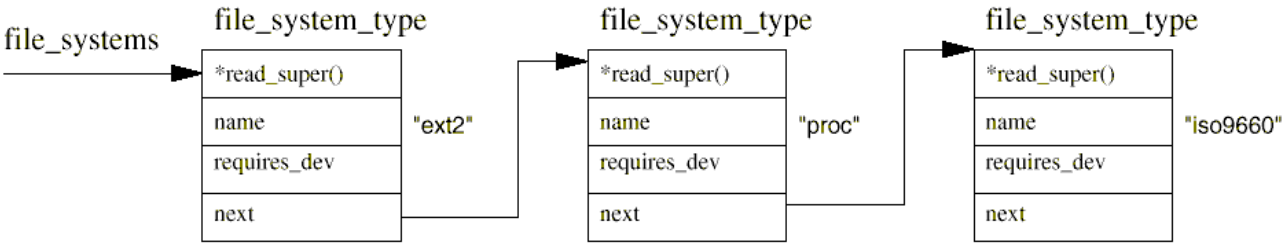


그림 9.5 : 등록된 파일 시스템

9.2.3 파일 시스템 등록하기

fs/filesystem.c sys_setup() 참조

include/linux/fs.h file_system_type 참조

리눅스 커널을 빌드할 때 어떤 파일 시스템을 지원할 것인지 지정할 수 있다. 커널을 빌드할 때, 파일 시스템 시작 코드는 내장된 모든 파일 시스템의 초기화 루틴을 호출한다. 리눅스 파일 시스템은 모듈로 만들어질 수도 있는데, 이 경우에는 필요할 때 로드되거나, insmod에 의해 수작업으로 로드된다. 파일 시스템 모듈은 로드될 때마다 자신을 커널에 등록하고, 언로드될 때 자신을 해제한다. 각 파일 시스템의 초기화 루틴은 자신을 가상 파일 시스템(VFS)에 등록하며, `file_system_type` 자료구조에 의해 표현된다. 자료구조에는 파일 시스템의 이름과 VFS 수퍼블럭 읽기 루틴에 대한 포인터가 저장되어 있다. 그림 9.5는 `file_system_type` 자료구조가 `file_systems` 포인터가 가리키는 리스트로 저장되어 있는 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 시스템이 저장될 장치가 필요한 것은 아니다. 예를 들어, `/proc` 파일 시스템은 블럭 장치가 필요로 하지 않는다.

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

```
       ext2
nodev  proc
       iso9660
```

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 포함하고 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

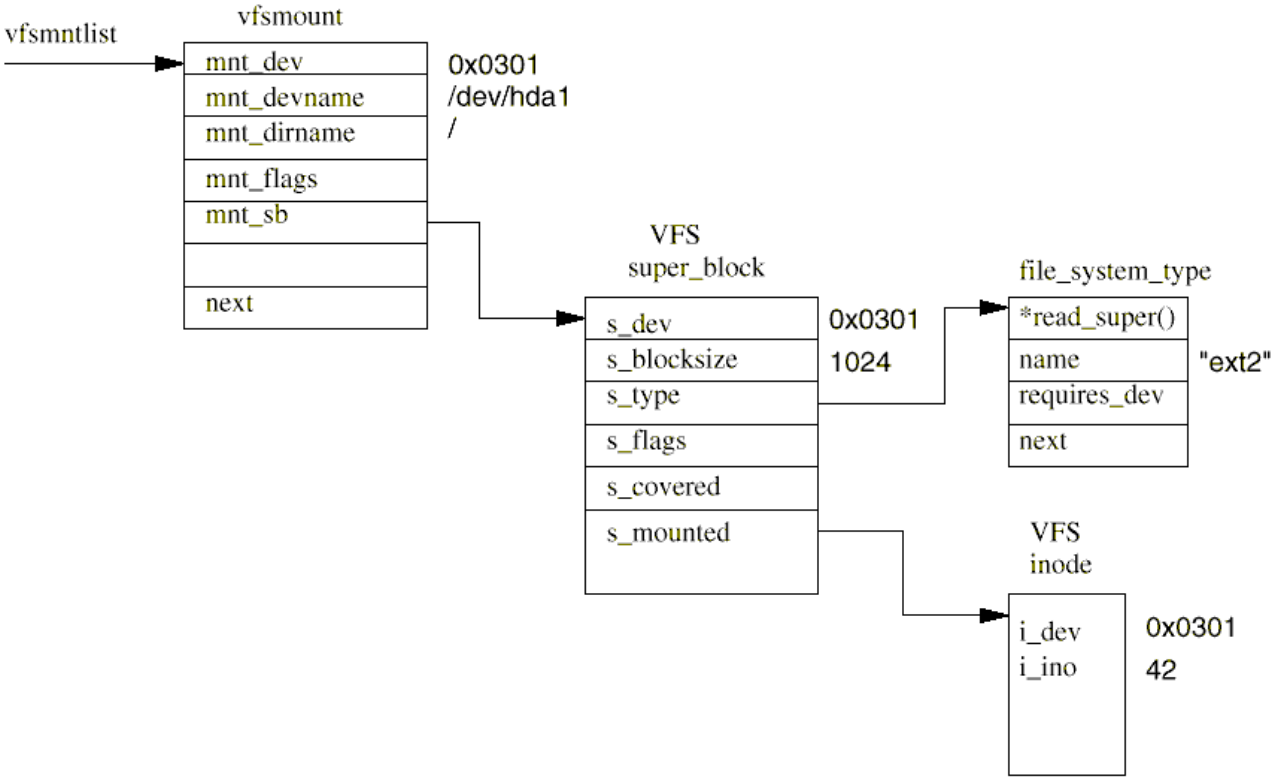


그림 9.6 : 마운트된 파일 시스템

fs/super.c
do_mount() 참조

fs/super.c
get_fs_type()
참조

알려진 파일 시스템들의 리스트를 탐색한다. 즉 `file_systems`가 가리키는 리스트에서 각각의 `file_system_type` 자료구조를 살펴본다. 일치하는 이름을 찾는다면, 이 파일 시스템 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 것은 아니다. 이 경우에는 커널은 작업을 계속하기 전에 커널 데몬이 적절한 파일 시스템 모듈을 로드하도록 요청한다.

다음으로 만약 `mount`에 전달된 물리적 장치가 아직 마운트되지 않았다면, 새로운 파일 시스템의 마운트 지점이 될 디렉토리의 VFS inode를 찾아야 한다. 이 VFS inode는 inode 캐시에 있거나, 아니면 마운트 지점의 파일 시스템을 저장하고 있는 블럭 장치에서 읽어야 한다. inode를 찾으면 이것이 디렉토리인지, 그리고 여기에 이미 다른 파일 시스템이 마운트된 것은 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 함께 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 수퍼블럭을 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 항목을 가리키고, mru_vfsmnt 포인터는 가장 최근에 사용된 파일 시스템을 가리킨다. 각 vfsmount 구조는 파일 시스템을 담고있는 블럭 장치의 장치 번호, 이 파일 시스템이 마운트된 디렉토리, 이 파일 시스템이 마운트될 때 할당된 VFS 수퍼블럭에 대한 포인터 등을 갖고 있다. VFS 수퍼블럭은 해당 종류의 파일 시스템에 대한 file_system_type 자료구조와 이 파일 시스템의 루트 inode를 가리킨다. 이 inode는 이 파일 시스템이 로드되어 있는 동안 VFS inode 캐시에 항상 존재한다.

fs/super.c add_vfsmnt() 참조

9.2.5 가상 파일 시스템(VFS)에서 파일 찾기

가상 파일 시스템에서 어떤 파일의 VFS inode를 찾으려면, VFS는 파일 이름을 구성하는 중간 디렉토리를 나타내는 VFS inode를 한번에 하나씩 찾아가며 디렉토리 이름을 해석해야 한다. 각 디렉토리를 검색하는 과정에서 파일 시스템 고유의 검색 함수를 호출하게 되며, 이 함수의 주소는 부모 디렉토리를 나타내는 VFS inode에 저장되어 있다. 이것이 가능한 이유는 항상 각 파일 시스템의 루트의 VFS inode를 알고 있고, 이것은 파일 시스템의 VFS 수퍼블럭에서 가리키고 있기 때문이다. 실제 파일 시스템이 어떤 inode를 검색할 때, 각 디렉토리에 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 어느정도 적용된다. 파일 시스템의 마운트를 해제하려면 시스템에서 그 파일 시스템내의 파일을 사용하고 있는 것이 없어야 한다. 따라서 어떤 프로세스가 /mnt/cdrom 디렉토리나 그 아래 디렉토리를 사용하고 있다면 마운트를 해제할 수 없다. 만약 무엇인가가 마운트를 해제하려는 파일 시스템을 사용하고 있다면, VFS inode 캐시에 그 파일 시스템에 속하는 VFS inode가 들어 있을 것이다. 따라서 마운트 해제 프로그램은 해제하려는 파일 시스템이 차지하고 있는 장치에 속하는 inode가 캐시의 inode 리스트에 들어 있는지 검사한다. 마운트된 파일 시스템의 VFS 수퍼블럭이 더티하면, 즉 내용이 수정되었다면, 수퍼블럭을 디스크의 파일 시스템에 기록하여야만 한다. 일단 디스크에 기록하고 나면 VFS 수퍼블럭이 차지하고 있던 메모리를 커널의 메모리 풀에 보내준다. 그리고 마지막으로 이 파일 시스템의 마운트에 필요했던 vfsmount라는 데이터 구조를 vfsmntlist로 부터 떼어낸 다음 해제한다.

fs/super.c do_umount() 참조

fs/super.c remove_vfsmnt() 참조

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 유지한다. VFS inode를 inode 캐시에서 읽을 수 있다면 그만큼 실제 장치에 대한 액세스를 덜 해도 된다.

fs/inode.c 참조

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 시스템을 갖고 있는 실제 장치의 장치 식별자로부터 계산된다. 가상 파일 시스템이 inode를 액세스할 필요가 있을 때 마다 VFS inode 캐시를 먼저 찾아본다. 캐시내의 inode를 찾기 위해서 시스템은 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 것과 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 파일 시스템이 메모리로부터 inode를 읽을 수 있도록 빈 VFS inode를 찾아야만 한다. VFS가 빈 inode를 찾는데에는 여러가지 방법이 있다. 만약 시스템이 VFS inode를 더 할당할 수 있다면 다음과 같은 방법을 쓰게 된다 - 커널 페이지를 할당하고 이를 여러 개의 새로운 빈 inode로 쪼갠다음 inode 리스트에 넣는다. 시스템에 있는 모든 VFS inode는 `first_inode`가 가리키는 리스트와 inode 해시 테이블에 들어있게 된다. 만약 시스템에 허용된 만큼 모든 inode를 이미 할당하였다면 재사용할만한 inode 후보들을 찾아야만 한다. 사용 횟수가 0인 inode는 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 재사용의 후보가 되는 경우가 결코 없다. 일단 재사용 후보가 선택되면 그 내용을 깨끗이 지운다. 만약 VFS inode가 더티하면 파일 시스템에 그 내용을 기록할 필요가 있으며, 만약 락이 되어 있다면 락이 풀릴 때까지 기다려야 한다. 후보 VFS inode는 재사용되기 전에 반드시 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 1이 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 있다. 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)

fs/dcache.c 참조

흔히 쓰이는 디렉토리에 대한 액세스 속도를 높이기 위해, VFS에서는 디렉토리 엔트리에 대한 캐시를 유지한다. 디렉토리는 실제 파일 시스템에 의하여 참조되므로 실제 파일 시스템에 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 실행중이라면 `/usr/X11R6/bin` 디렉토리는 매우 자주 액세스될 것이다.

디렉토리 캐시는 해시 테이블로 구성되는데, 이 테이블에서 각각의 엔트리는 같은 해시 값을 가진 디렉토리 캐시 엔트리들의 리스트를 가리키고 있다. 해시 함수는 파일 시스템을 갖고 있는 장치의 장치 번호와 디렉토리 이름을 이용하여 해시 테이블 내의 위치 즉 인덱스를 산출해낸다. 이렇게 함으로써 캐시된 디렉토리 엔트리를 빨리 찾을 수 있다. 엔트리를 찾는데 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 사용된, Least Recently Used) 방식으로 디렉토리 캐시 엔트리 리스트를 관리한다. 디렉토리 엔트리는 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 2단계의 LRU 리스트에서 맨 앞의 엔트리를 대치하는 것은 제대로 된 것이다. 어떤 엔트리가 리스트의 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 안전하다. 엔트리가 2단계 리스트에 들어있다는 것은 액세스되었을 뿐만 아니라 반복적으로

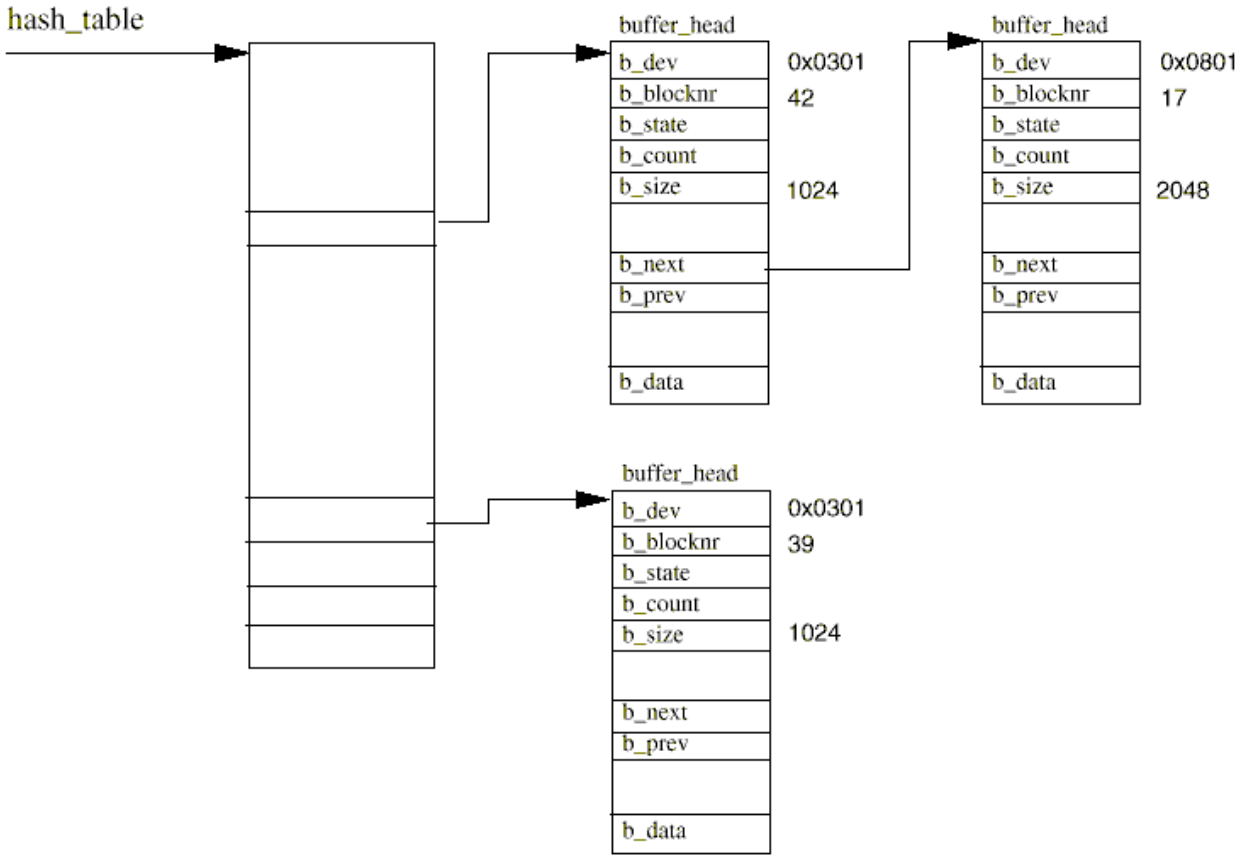


그림 9.7 : 버퍼 캐시

참조되고 있음을 의미하기 때문에 더 안전하게 보관할 필요가 있다.

REVIEW NOTE : 그림이 필요한가?

9.3 버퍼 캐시(Buffer Cache)

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

캐시에 있는 블럭 버퍼는 이를 소유하는 장치 식별자와 버퍼의 블럭번호로 유일하게 구별된다. 버퍼 캐시는 두개의 기능적인 부분으로 되어있다. 첫번째 부분은 프리 블럭 버퍼의 리스트이다. 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 인덱스를 가지는 버퍼들을 고리로 가리키고 있는 포인터들의 벡터이다. 해시 인덱스는 해당 장치 식별자와 데이터 블럭의 블럭 번호로부터 만들어진다. 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 버퍼를 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 캐시로부터 블럭을 얻으려고 시도한다. 만약 버퍼 캐시에서 버퍼를 얻을 수 없다면, 프리 리스트에서 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 있다. 만약 최근 것이 아니거나, 새 블럭 버퍼라면, 파일 시스템은 디바이스 드라이버에게 해당하는 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

fs/buffer.c
bdflush() 참조

`bdflush` 커널 데몬은 시스템이 너무 많은 더티 버퍼 - 언젠가는 디스크에 쓰여져야 하는 데이터를 가지고 있는 버퍼 - 를 가지게 되었을 때 동적으로 반응하는 간단한 커널 데몬이다. 이는 시스템이 시작할 때 커널 쓰레드로서 시작되며, 혼동되지 않도록 자신을 kflushd라고 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 기다리며 잠들어있다. 버퍼가 할당되거나 버려질 때 시스템에 있는 더티 버퍼의 갯수를 검사한다. 만약 시스템에 있는 전체 버퍼의 갯수 중에서 더티 버퍼의 비율이 너무 커지면 `bdflush`가 깨어난다. 기본값으로 설정된 값은 60%이지만, 시스템에서 버퍼가 필요하다면 `bdflush`는 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 연결되고, bdflush는 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 초기화동안에), 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.

fs/buffer.c sys_bdflush() 참조

9.4 /proc 파일 시스템

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

/proc 파일 시스템은 사용자에게 커널의 내부 작업을 볼 수 있는 있는 창을 제공한다. 12장에서 설명하고 있는 리눅스 커널 모듈같은 어떤 리눅스 서브시스템들은 /proc 파일 시스템에 엔트리를 생성하기도 한다[99].

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 영역도 차지하지 않는다. 이는 단지 디바이스 드라이버로의 접근점일 뿐이다. EXT2 파일 시스템과 리눅스 VFS는 모두 장치 파일을 inode의 특수한 유형으로 구현한다. 장치 파일에는 문자 특수 파일과 블럭 특수 파일이라는 두가지 형태가 있다. 커널 안에서, 디바이스 드라이버는 파일처럼 구현되어 있다. 즉, 이를 열고, 닫는 등의 일을 할 수 있다. 문자 장치는 문자모드로 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 디바이스 드라이버로 전달된다. 종종 이는 실제 디바이스 드라이버가 아니라, SCSI 디바이스 드라이버 계층과 같은 어떤 서브 시스템을 위한 유사 디바이스 드라이버이기도 한다. 장치 파일은 장치의 유형을 구별하는 메이저 번호와, 한 덩어리 또는 그 메이저 유형의 한 사례를 구별하기 위한 마이너 유형으로 참조한다. 예를 들어, 첫번째 시스템에서 IDE 컨트롤러에 있는 IDE 디스크들은 메이저 번호로 3을 가지며, IDE 디스크의 첫번째 파티션은 마이너 번호로 1을 가진다. 따라서 ls -l /dev/hda1을 하면 다음과 같은 출력을 보여준다.

include/linux/major.h 에서 리눅스의 모든 메이저 장치 번호를 볼 수 있다.

역주 99) 디바이스 드라이버를 포함하여 다른 커널 부분도 proc_register_dynamic() 함수에 적절한 인자를 전달하고, 파일 연산을 수행할 수 있는 함수를 구현함으로써 proc 파일 시스템에 엔트리를 만들 수 있다. (flyduck)

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

include/linux/kdev_t.h 참조

커널에서, 모든 장치는 `kdev_t` 자료형으로 유일하게 표현된다. 이는 2바이트 길이로 첫번째 바이트는 마이너 장치 번호를, 두번째 바이트는 메이저 장치 번호를 갖는다[100]. 위에 보여준 IDE 장치는 커널에서 *0x0301*을 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 올바른 장치 식별자로 설정한다.

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

역주 100) 유닉스에서 전통적으로 장치의 번호를 간직하는데 `dev_t`라는 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 일으킬 수 있다. 그래서 리눅스는 장치 번호를 나타내는데 `kdev_t`라는 새로운 자료형을 선언하고 이를 사용하고 있다. 이 자료형은 메이저 번호와 마이너 번호가 각각 16비트의 크기를 갖는다. include/linux/kdev_t.h 참조 (flyduck)

10 장

네트워크 (Networks)

네트워킹과 리눅스는 거의 동의어이다. 리눅스는 말 그대로 인터넷 또는 월드 와이드 웹 (World Wide Web, WWW)의 산물이다. 리눅스의 개발자와 사용자들은 정보와 프로그램 코드을 교환하기 위해 웹을 사용하며, 조직의 네트워킹 요구를 처리하기 위해 리눅스를 자주 사용한다[101]. 이 장은 리눅스가 통틀어 TCP/IP 라고 부르는 네트워크 프로토콜을 어떻게 지원하는지 설명한다.

TCP/IP 는 미국 정부가 출자하는 미국 연구망(ARPANET)에 연결된 컴퓨터 간의 통신을 지원하기 위해 구상된 것이다. ARPANET 은 패킷 스위칭과 하나의 프로토콜이 다른 프로토콜의 서비스을 사용하는 프로토콜 계층화 등의 네트워킹 개념을 창시했다. ARPANET 은 1988 년에 종료되었지만 그 계승자인 NSF[102] NET 과 인터넷은 더 크게 성장했다. 현재 월드 와이드 웹이라고 알려진 것은 ARPANET 으로부터 성장했으며, TCP/IP 프로토콜을 바탕으로 하고 있다. ARPANET 상에서는 유닉스가 광범위하게 사용되었으며, 처음으로 네트워킹이 가능한 유닉스 버전은 4.3 BSD 였다. 리눅스의 네트워킹 구현은 4.3 BSD 를 모델로 설계되었으며, 리눅스는 (약간 확장된) BSD 소켓과 TCP/IP 네트워킹 전체를 지원한다. 리눅스에서 이 TCP/IP 프로그래밍 인터페이스를 선택한 이유는 TCP/IP 가 널리 사용되고 있으며, 리눅스와 다른 유닉스 플랫폼과의 응용 프로그램 호환성을 높이기 위한 것이었다.

10.1 TCP/IP 네트워킹의 개관

이 절은 TCP/IP 네트워킹의 기본 원리에 대한 개관이다. 이것은 (이후의 절과 같은) 상세한 설명이 아니기 때문에 한번 읽어보기 바란다.

IP 네트워크에서는 각 기계를 고유하게 식별하는 32 비트 숫자인 IP 주소를 각 기계에 부여한다. WWW 는 매우 거대하고 계속 성장하는 IP 네트워크로서, WWW 에 연결된 모든 기계들은 할당된 고유한 IP 주소를 가진다. IP 주소는 예를 들어 16.42.0.9 와 같이 점으로 구분되는 네 개의 숫자로 나타낸다. 실제로는 네트워크 주소와 호스트 주소의 두 부분으로 IP 주소를 구분한다. (IP 주소에는 여러 클래스들이 있어서) 각 부분의 크기는 달라질 수 있지만, 16.42.0.9 를 예로 들면 16.42 는 네트워크 주소이고 0.9 는 호스트 주소가 된다. 호스트 주소는 서브네트워크와 호스트 주소로 더 (자세히) 나눌 수 있다. 다시 16.42.0.9 를 예로 들면, 서브네트워크 주소는 16.42.0 이 되고 호스트 주소는 16.42.0.9 가 된다. 이렇게 IP 주소를 몇 구획으로 나눌 수 있으므로, (네트워크를 사용하는) 기관은 자신의 네트워크를 몇 구획으로 나눌 수 있다. 예를 들어 16.42 가 ACME 컴퓨터사의 네트워크 주소라면, 16.42.0 는 서브네트

역주 [101]) 가장 널리 사용되는 웹 서버인 아파치의 절반 이상이 리눅스에서 동작중이다. (심마로)

[102]) 국립 과학 재단(National Science Foundation)

워크 0 번, 16.42.1 은 서브네트워크 1 번이 될 것이다. 이 서브네트워크는 서로 다른 건물에 있을 수도 있고, 임대 전화선을 이용하거나 무선(통신수단)을 이용해 연결되어 있을 수도 있다. IP 주소는 네트워크 관리자가 할당하는데, IP 서브네트워크를 사용하여 네트워크 관리 부담을 분산시킬 수 있다. IP 서브네트워크 관리자는 자신의 IP 서브네트워크 안에서 자유롭게 IP 주소를 할당할 수 있다.

하지만 일반적으로 IP 주소는 아주 기억하기 어렵다. 이름을 붙이는 것이 훨씬 (기억하기) 쉽다. `linux.acme.com` 이 16.42.0.9 보다 훨씬 더 기억하기 쉬운데, (이름을 사용하기 위해서는) 네트워크 이름을 IP 주소로 변환해 주는 도구가 필요하다. 이 이름들을 `/etc/hosts` 파일에 정적으로 명시할 수도 있지만, 리눅스는 분산 네임 서버(Distributed Name Server, DNS)에 이 이름들을 변환해 달라고 요청할 수도 있다. 이 경우 로컬 호스트는 하나 이상의 DNS 서버의 IP 주소를 알고 있어야만 하는데, 이 주소들을 `/etc/resolv.conf` 에 기록한다.

웹 페이지를 읽을 때와 같이 다른 기계에 접속할 때마다 그 기계와 자료를 교환하기 위해 그 기계의 IP 주소를 사용한다. 자료들은 IP 패킷에 담겨 전달되는데, 각 패킷마다 출발지 기계와 목적지 기계의 IP 주소, 체크섬(checksum) 및 다른 유용한 정보를 담고 있는 IP 헤더가 붙어 있다. 체크섬은 IP 패킷에 있는 데이터를 가지고 계산하는데, 이를 이용하여 IP 패킷 수신자는 전화선의 잡음 등으로 인해 전달과정에서 패킷이 손상되었는지를 판단할 수 있다. 응용 프로그램이 보내는 데이터는 좀 더 다루기 쉬운 작은 패킷들로 쪼개질 수 있다. IP 데이터 패킷의 크기는 연결 매체에 따라 달라지는데, 일반적으로 이더넷 패킷이 PPP 패킷보다 더 크다. 목적지 호스트는 데이터 패킷들을 다시 조합하여 응용 프로그램에 데이터를 건내준다. 느린 시리얼 링크를 통해 많은 그래픽 이미지들을 담고 있는 웹 페이지를 보면 위에서 말한 데이터의 분해와 조립 과정을 그림을 보듯 살펴볼 수 있다.

같은 IP 서브네트워크에 연결되어 있는 호스트끼리는 IP 패킷을 직접 보낼 수 있지만, 그렇지 않은 경우에는 게이트웨이(gateway)라고 하는 특별한 호스트에 IP 패킷을 보내야만 한다. 게이트웨이(또는 라우터)는 하나 이상의 IP 서브네트워크에 연결되어 있는데, 한 IP 서브네트워크에서 받은 패킷을 다른 IP 서브넷으로 전송한다. 예를 들어, 서브네트워크 16.42.1.0 과 16.42.0.0 이 어떤 게이트웨이를 통해 연결되어 있다면 서브네트워크 0 에서 서브네트워크 1 로 전달되는 패킷들은 게이트웨이로 보내지고 게이트웨이는 이 패킷을 전달한다. 각 호스트들은 정확한 기계에 IP 패킷을 전달하기 위해 라우팅 테이블(routing table)을 작성한다. 라우팅 테이블에는 모든 IP 목적지에 대해 그 목적지에 도달하기 위해 어떤 호스트에 IP 패킷을 전달해야 하는지를 결정하기 위해 사용되는 정보가 있다. 이 라우팅 테이블은 동적이어서 응용 프로그램이 네트워크를 사용하거나 네트워크 구성도가 변경되거나 하면 시간이 지남에 따라 변경된다.

IP 프로토콜은 다른 프로토콜이 데이터를 보낼 때 사용하는 전송 계층이다. TCP 는 신뢰할 수 있는 일대일 프로토콜로서, 데이터를 주고 받기 위해 IP 프로토콜을 사용한다. IP 패킷에 헤더가 붙어 있는 것처럼, TCP 패킷에도 헤더가 붙어 있다. TCP 는 연결 중심적인 프로토콜로 (이를 사용하는) 두 네트워크 응용 프로그램은 그 사이에 많은 서브네트워크, 게이트웨이 및 라우터가 있더라도 단일한 가상의 접속을 통해 연결된다. TCP 는 두 응용프로그램간의 데이터를 신뢰할 수 있는 방식으로 전달하며 데이터의 손실이나 중복이 없다는 것을 보장한다. TCP 가 IP 를 사용하여 TCP 패킷을 전송할 때, IP 패킷에 들어있는 데이터는 바로 TCP 패킷이다. 서로 통신하고 있는 호스트의 IP 계층은 IP 패킷을 주고 받는 역할을 한다. UDP 도 (UDP) 패킷을 전달하는데 IP 계층을 사용하지만, TCP 와는 달리 UDP 는 신뢰할 수 없는 프로토콜이며 데이터그램(datagram) 서비스를 제공한다. 이와 같이 다른 프로토콜이 IP 를 사용하려면, IP 패킷을 받을 때 IP 계층이 이 IP 패킷에 담긴 데이터를 어떤 상위 프로토콜에 전달해야 하는지를 알고 있어야만 한다. 이를 위해 모든 IP 패킷 헤더에는 프로토콜 식별자를 지정하는 바이트가 있다. TCP 가 IP 계층에다 IP 패킷을 전송하도록 요청하면, 그 패킷에 TCP 패킷이 들어있다는 것을 IP 패킷 헤더에 기록한다. IP 계층이 데이터를 받으면, 이 프로토콜 식별자를 사용하여, 받은 데이터를 상위의 어떤 계층에 전달할지를 결정한다. 이 경우에는 TCP 계층이 될 것이다. 응용프로그램이 TCP/IP 를 통해 통신을 할 때, 응용프로그램은 상대방의 IP 주소뿐만 아니라 그 응용프로그램의 포트 주소 또한 명시하여야 한다. 포트 번호는 응용프로그램마다 유일하며, 표준 네트워크 응용프로그램은 표준 포트번호를 사용한다.

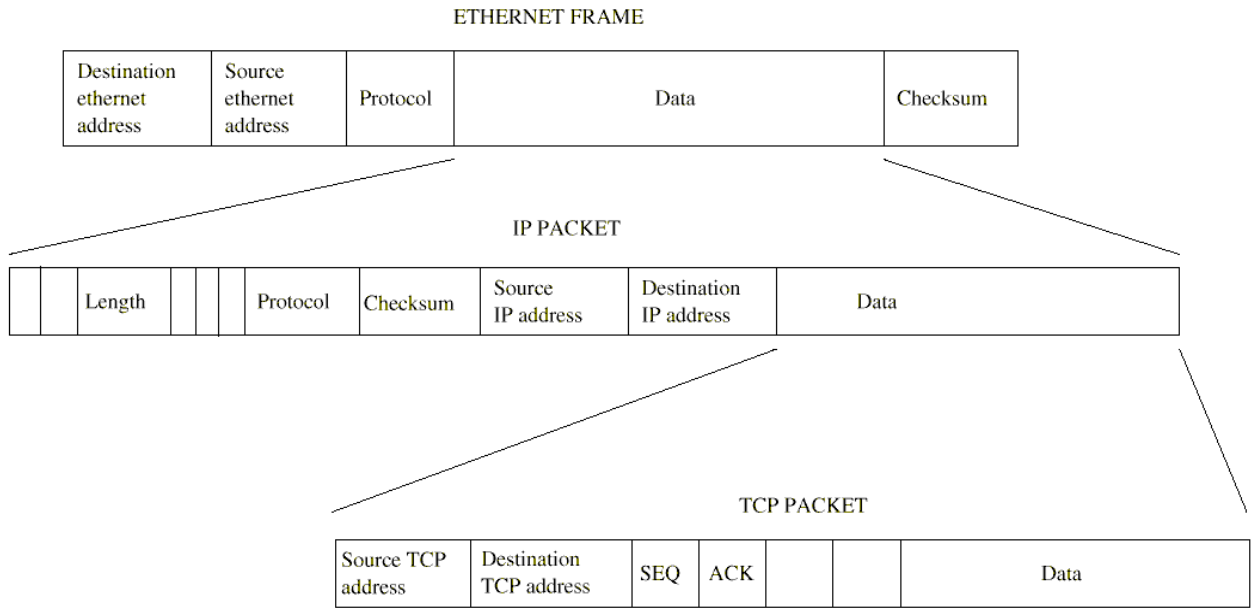


그림 10.1 : TCP/IP 프로토콜 계층

예를 들어, 웹서버는 80 번 포트를 사용한다. 이러한 등록된 포트번호는 `/etc/services` 에서 볼 수 있다.

프로토콜의 계층구조는 TCP, UDP 및 IP 로 (구분하는 것으로) 끝나는 것이 아니다. IP 프로토콜 자체도 IP 패킷을 다른 IP 호스트로 전송하는데 수많은 장치들을 사용한다. 이 장치는 자신만의 프로토콜 헤더를 추가하기도 한다. 이러한 예로는 이더넷 계층이 있으며, 또 다른 예로 PPP 와 SLIP 이 있다. 이더넷 네트워크에서 많은 호스트가 실제 케이블 하나에 동시에 접속할 수 있다. 전송되는 모든 이더넷 프레임은 연결된 모든 호스트에 보이게 되므로[103] 모든 이더넷 장치는 고유한 주소를 갖는다. 호스트는 자기 주소로 배달되는 모든 이더넷 프레임을 받아들이지만, 같은 네트워크에 연결된 다른 호스트들은 이를 무시하게 된다. 이더넷의 이런 유일한 주소는 이더넷 장치를 만들 때 적어넣게 되는데, 일반적으로 이더넷 카드의 SROM[104]에 들어 있다. 이더넷 주소는 6 바이트 길이인데 예를 들면 08-00-2B-00-49-A4 같은 값을 갖는다. 어떤 이더넷 주소는 멀티캐스트(multicast) 목적으로 예약되어 있는데, 이런 주소로 보내지는 이더넷 프레임은 같은 네트워크 안에 있는 모든 호스트가 받는다. 이더넷 프레임은 (데이터로) 수많은 프로토콜들을 전송할 수 있기 때문에, IP 패킷과 같이 헤더에 프로토콜 식별자가 있다. 이에 따라 이더넷 계층은 정확하게 IP 패킷을 받아 IP 계층에 전달할 수 있다.

이더넷과 같은 다중 접속 프로토콜을 통해 IP 패킷을 보내기 위해서는 IP 계층은 IP 호스트의 이더넷 주소를 찾아야만 한다. IP 어드레스는 단지 개념적인 주소일 뿐이고, 고유한 물리적인 주소를 가지고 있는 것은 이더넷 장치이기 때문이다. 반면에 IP 주소는 네트워크 관리자의 의지대로 지정되고 변경될 수 있지만, 네트워크 하드웨어는 자신의 물리적 주소 또는 모든 기계가 받아야만 하는 특별한 멀티캐스트에만 반응한다. 리눅스는 IP 주소를 이더넷 주소와 같은 실제 하드웨어 주소 변환하기 위해 ARP(Address Resolution Protocol)를 사용한다. 특정한 IP 주소를 가진 하드웨어 주소를 알고자 하는 호스트는 변환하고자 하는 IP 주소가 담긴 ARP 요청 패킷을 멀티캐스트 주소에 보내 모든 노드에 전달한다. 그 IP 주소를 가지고 있는 호스트는 자신의 하드웨어 주소가 담긴 ARP 응답을 돌려준다. ARP 는 이더넷 장치만 사용되는 것이 아니라 IP 주소를 FDDI 와 같은 다른 물리적 장치의 주소로 변화하는데에

역주 [103]) 이더넷은 방송 프로토콜을 사용하고, 이 때문에 보안성이 떨어지는 측면이 있다 (심마로)

[104]) 동기적 읽기 전용 메모리(Synchronous Read Only Memory)

도 사용할 수 있다. ARP 를 할 수 없는 네트워크 장치들은 따로 표시를 해 두어 리눅스는 (이 장치에 대해서는) ARP 를 시도하지 않는다. 이와는 반대되는 기능으로 RARP(Reverse Address Resolution Protocol)가 있는데, 이것은 물리적 네트워크 주소를 IP 주소로 변환한다. 이 기능은 게이트웨이가 사용하는데, 게이트웨이는 원격 네트워크에 있는 IP 주소를 대신해서 ARP 요청에 응답한다.

10.2 리눅스의 TCP/IP 네트워킹 계층

네트워크 프로토콜과 마찬가지로, 그림 10.2 에서 볼 수 있는 것처럼 리눅스는 인터넷 프로토콜 주소 패밀리(address family)를 일련의 연관된 소프트웨어 계층으로 구현하고 있다. BSD 소켓은 BSD 소켓만 처리하는 일반적인 소켓 관리 소프트웨어가 지원한다. INET 소켓 계층은 소켓 관리 소프트웨어를 지원하는데, 이것은 IP 기반의 프로토콜인 TCP 와 UDP 의 통신 종점을 관리한다. UDP(User Datagram Protocol)는 비연결지향 방식의 프로토콜(connectionless protocol)인데 비해, TCP(Transmission Control Protocol)는 연결지향의 신뢰할 수 있는 일대일 프로토콜이다. UDP 패킷을 전송할 때, 리눅스는 그 패킷이 목적지에 안전하게 도착하였는지를 알 수도 없고 신경을 쓰지도 않는다. TCP 패킷들에는 번호를 매겨, TCP 접속의 양 끝(종점 호스트)은 전송 데이터가 정확하게 수신되었는지를 확인한다. IP 계층에는 인터넷 프로토콜을 구현한 코드가 들어 있다. 이 코드는 전송하는 데이터 앞에 IP 헤더를 붙이고, 들어오는 IP 패킷을 TCP 나 UDP 계층으로 어떻게 전달하는지를 알고 있다. IP 계층 아래에서 PPP 또는 이더넷과 같은 네트워크 장치들이 리눅스의 모든 네트워킹을 지원한다. 네트워크 장치라고 항상 물리적인 장치만을 가리키는 것은 아니다. 루프백 장치와 같은 몇몇 장치는 순전히 소프트웨어로만 작성되어 있다. `mknod` 명령으로 만들어지는 표준적인 리눅스 장치와는 달리, 네트워크 장치는 관련된 소프트웨어가 장치를 찾아내 초기화해야지만 나타난다. 그래서 해당하는 이더넷 디바이스 드라이버를 넣어서 커널을 빌드해야만 `/dev/eth0` 를 볼 수 있다. ARP 프로토콜은 IP 계층과 각종 주소에 대한 ARP 를 지원하는 프로토콜 사이에 있다.

10.3 BSD 소켓 인터페이스(Socket Interface)

BSD 소켓 인터페이스는 다양한 형태의 네트워킹 뿐만 아니라 프로세스간 통신도 지원하는 일반적인 인터페이스이다. 소켓은 통신 연결의 한쪽 끝으로 생각할 수 있는데, 통신하고 있는 두 프로세스는 통신 연결에서 자신쪽 끝에 해당하는 소켓을 가지게 된다. 소켓을 특별한 종류의 파이프로 생각할 수도 있지만, 파이프와는 달리 소켓은 거기에 담을 수 있는 데이터의 양에 제한이 없다. 리눅스는 몇 가지 클래스의 소켓을 지원하는데, 이것들을 주소 패밀리(address family)라고 부른다. 이는 각 클래스별로 자신의 통신에 사용하는 주소 표현법을 가지고 있기 때문이다. 리눅스는 다음과 같은 소켓 주소 패밀리 또는 도메인을 지원한다.

UNIX	유닉스 도메인 소켓 (Unix domain `socket`)
INET	TCP/IP 프로토콜을 이용한 통신을 지원하는 인터넷 주소 패밀리
AX25	아마추어 라디오 X.25
IPX	노벨의 IPX 프로토콜
APPLETALK	애플사의 Appletalk DDP 프로토콜
X25	X.25 프로토콜

소켓에는 몇가지 타입이 있으며, 이는 접속을 지원하는 서비스의 종류를 나타낸다. 모든 주소 패밀리가 모든 형태의 서비스를 지원하는 것은 아니다. 리눅스 BSD 소켓은 몇가지 소켓 타입을 지원한다.

스트림(Stream) 이 소켓은 데이터가 전송 중 분실, 오염 또는 중복되지 않는다는 것을 보장하는 신뢰할 수 있는 양방향 순차 데이터 스트림을 제공한다. INET 주소 패밀리의 TCP 프로토콜이 스트림 소켓을 지원한다.

데이터그램(Datagram) 이 소켓은 양방향 데이터 전송을 제공하지만, 스트림 소켓과는 달리

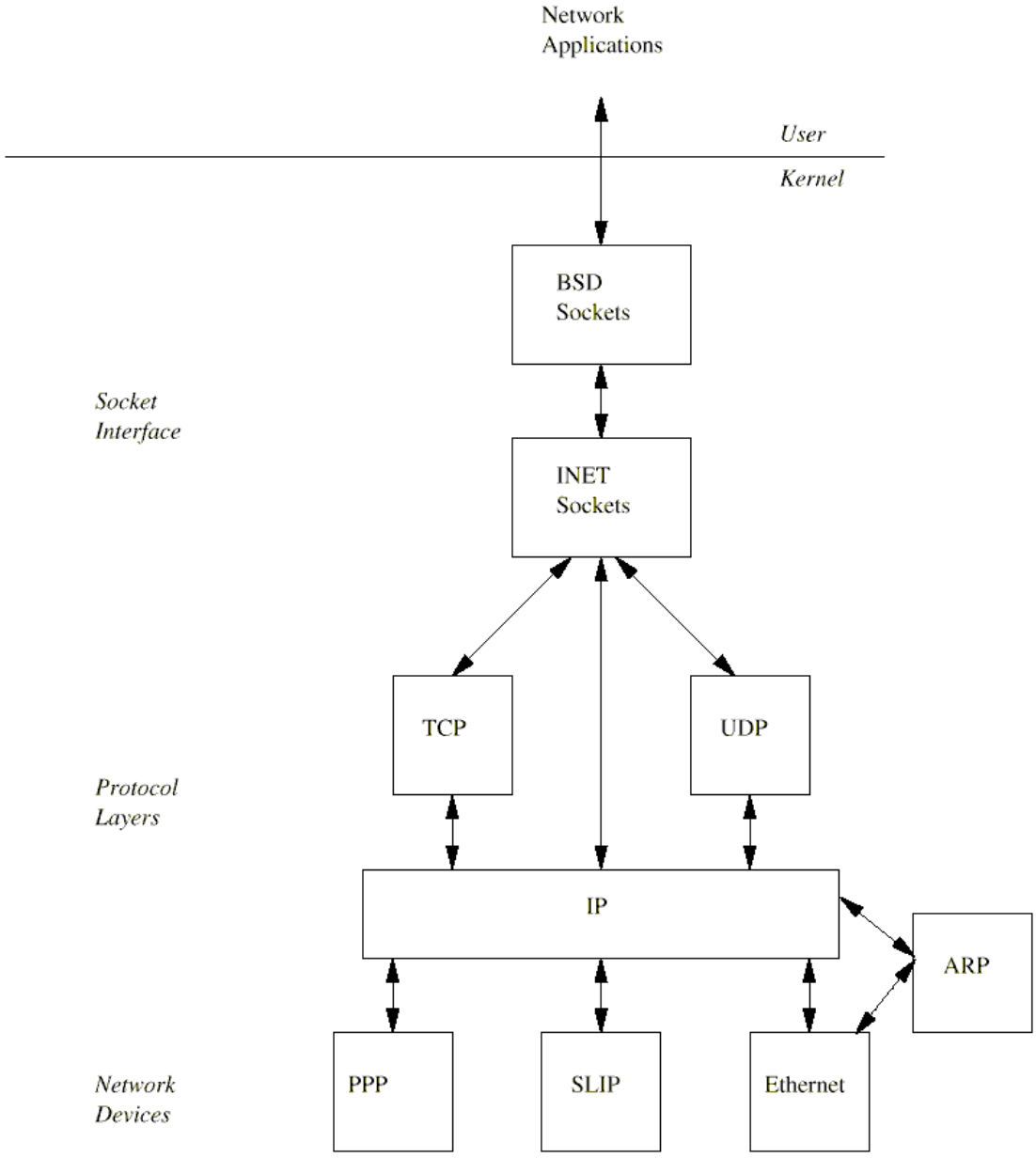


그림 10.2 : 리눅스 네트워킹 계층

그 메시지가 (제대로) 도착한다는 것을 보장하지는 않는다. 메시지가 목적지에 도착하였다 하더라도, 메시지가 순서에 맞게 또는 중복되거나 오염되지 않고 도착하였다는 것을 보장하지 않는다. INET 주소 패밀리의 UDP 프로토콜이 이 종류의 소켓을 지원한다.

가공하지 않은(Raw) 프로세스가 하부 프토토콜에 직접 접근(그래서 "raw")할 수 있는 소켓이다. 예를 들면 이더넷 장치에 이 소켓을 열어 가공되지 않은 IP 데이터 흐름을 지켜보는 것이 가능하다.

도착 신뢰 메시지(Reliable Delivered Messages) 이것은 데이터그램 소켓과 아주 비슷하지만 데이터가 (목적지에) 도착한다는 것을 보장한다.

순차적 패킷(Sequenced Packets) 이것은 스트림 소켓과 비슷한데 데이터 패킷의 크기가 고정되어 있다.

패킷(Packet) 이것은 표준 BSD 소켓 타입은 아니고, 장치 수준에서 프로세스가 직접 패킷에 접근할 수 있는 리눅스 특유의 확장이다.

소켓을 사용하여 통신을 하는 프로세스는 클라이언트 서버 모델을 따른다. 서버는 서비스를 제공하고 클라이언트는 이 서비스를 이용한다. 이런 예로 웹 페이지를 제공하는 웹 서버와 그 페이지들을 읽는 웹 클라이언트 또는 브라우저를 들 수 있다. 소켓을 사용하는 서버는 먼저 소켓을 만든 후 소켓에 이름을 바인드(bind)한다. 이 이름의 형식은 소켓의 주소 패밀리에 따라 달라지는데, 실제로는 서버의 로컬 주소가 된다. 소켓의 이름 또는 주소는 `sockaddr` 자료 구조를 이용해 명시한다. INET 소켓은 그것에 바인드된 IP 포트 주소를 가지게 된다. 등록된 포트 번호는 `/etc/services` 에서 볼 수 있다. 예를 들어, 웹 서버의 포트번호는 80 번이다. 소켓에 주소가 바인드되었다면, 서버는 그 바인드된 주소를 가리키는 연결 요청이 들어오는지 리슨(listen)을 한다. 연결 요청을 하는 클라이언트는 소켓을 만들고 서버의 주소를 명시하여 소켓에 대해 연결 요청을 한다. INET 소켓에서 서버의 주소는 서버의 IP 주소와 포트 번호이다. 이러한 연결 요청은 다양한 프로토콜 계층을 통해 전달되어 서버의 리슨 소켓에 도달하게 된다. 서버가 연결 요청을 받으면, 이것을 받아들이거나(accept) 또는 거부한다(reject). 연결 요청을 받아들이기로 하였다면, 서버는 연결을 받아들일 새로운 소켓을 만든다. 연결 요청을 리슨하는데 사용하는 소켓은 연결을 받아들이는데 사용할 수는 없다. 연결이 이루어지고 나면, 서버와 클라이언트는 자유롭게 데이터를 주고 받을 수 있다. 마지막으로, 연결이 더이상 필요없는 경우 소켓을 종료(shutdown)할 수 있다. 이 때 전송 중에 있는 데이터 패킷이 정확하게 처리되었는지에 유의하여야 한다.

BSD 소켓에 어떤 조작을 가하는 것이 무엇을 의미하는지는 어떤 주소 패밀리 위에서 작업을 하고 있느냐에 따라 다르다. TCP/IP 접속을 설정하는 것은 아마추어 라디오 X.25 접속을 설정하는 것과는 아주 다르다. 가상 파일 시스템과 마찬가지로 리눅스는 BSD 소켓 계층으로 소켓 인터페이스를 추상화한다. BSD 소켓 계층은 BSD 소켓 계층이 응용프로그램과 인터페이스하는 것에 관련된다. 이런 소켓 인터페이스는 독립된 주소 패밀리를 가지는 소프트웨어에 의해 지원을 받는다. 커널 초기화 과정에서, 커널에 구현된 주소 패밀리는 (자신이 지원하는) BSD 소켓 인터페이스와 함께 자신을 등록한다. 나중에 응용프로그램이 BSD 소켓을 만들고 사용할 때, BSD 소켓과 그것이 지원하는 주소 패밀리 사이의 연관이 만들어진다. 이러한 연관관계는 교차연결 자료구조와 주소 패밀리 고유의 지원 루틴 테이블을 통해 만들어진다. 예를 들어 응용프로그램이 새로운 소켓을 만들 때 BSD 소켓 인터페이스가 사용하는 주소 패밀리 고유의 소켓 생성 루틴이 있다.

커널을 설정할 때 (많은) 주소 패밀리와 프로토콜을 `protocols` 벡터에 넣는다. `protocols` 벡터에는 각 주소 패밀리 또는 프로토콜의 이름 (예를 들면 "INET")과 초기화 루틴이 들어간다. 시스템이 부팅되면서 소켓 인터페이스를 초기화할 때, 각 프로토콜의 초기화 루틴이 불리게 된다. 여기서 소켓 주소 패밀리 별로 일련의 프로토콜 연산 루틴을 등록하게 된다. 이것은 루틴들의 집합이며 각 루틴은 해당 주소 패밀리의 고유한 특정 연산을 수행한다. `proto_ops` 자료구조는 주소 패밀리 타입과 특정 주소 패밀리에 고유한 소켓 연산 루틴에 대한 포인터들의 집합으로 이루어져 있다. `pops` 벡터는 인터넷 주소 패밀리같은 (AF_INET 은 2 이다) 주소 패밀리 식별자로 인덱스 되어있다.

include/linux/net.h 참조

10.4 INET 소켓 계층

INET 소켓 계층은 TCP/IP 프로토콜들을 포함하는 인터넷 주소 패밀리를 지원한다. 위에서 설명한 것처럼 이들 프로토콜들은 계층적이고, 한 프로토콜이 다른 프로토콜의 서비스를 사용한다. 리눅스의 TCP/IP 코드와 자료구조는 이 계층구조를 반영한다. BSD 소켓 계층으로의 인터페이스는 네트워크 초기화 도중에 BSD 소켓 계층에 등록 한 인터넷 주소 패밀리 소켓 함수들을 통한다. 이들은 등록된 다른 주소 패밀리와 함께 `pops` 벡터에서 보관한다. BSD 소켓계층은 등록된 INET `proto_ops` 자료구조로부터 INET 계층의 소켓지원 루틴을 호출하여 필요한 일을 수행한다. 예를 들어, 주소 패밀리에 INET 을 주고 BSD 소켓을 만들라고 요구한다면, 이는 밑에 있는 INET 소켓 생성 함수를 사용하게 된다. BSD 소켓 계층은 이들 각각의 함수마다 INET 계층에 BSD 소켓을 나타내는 `socket` 자료구조를 전달한다. BSD `socket` 을 TCP/IP 에만 필요한 정보로 어지럽히기 보다는 INET 소켓 계층은 자신만의 자료구조인 `sock` 을 가지고 자신을 BSD `socket` 자료구조와 연결한다. 이런 연결은 그림 10.3

include/net/sock.h 참조

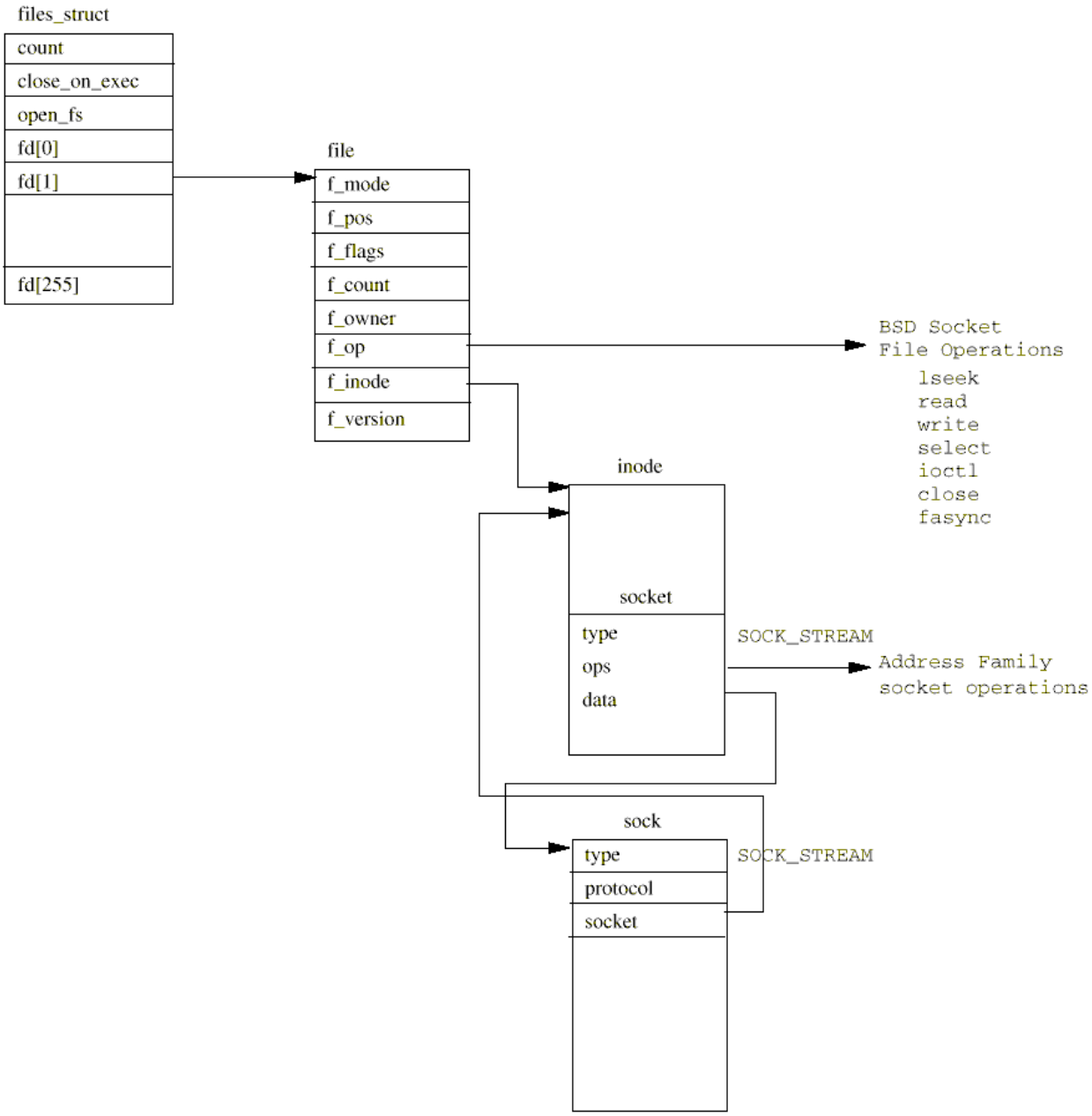


그림 10.3 : 리눅스 BSD 소켓 자료구조

에서 볼 수 있다. sock 자료구조는 BSD socket 에 있는 data 포인터를 통해 BSD socket 자료구조와 연결된다. 이것은 계속된 INET 소켓 호출에서 쉽게 sock 자료구조를 얻어올 수 있다는 의미이다. sock 자료구조의 프로토콜 함수 포인터 역시 생성시에 셋업이 되며, 이는 요구한 프로토콜에 따라 다르다. 만약 TCP 를 요구했다면, sock 자료구조의 프로토콜 함수 포인터는 TCP 연결을 위해 필요한 TCP 프로토콜 함수 집합을 가리킬 것이다.

10.4.1 BSD 소켓 만들기

net/socket.c
sys_socket()
참조

새 소켓을 만드는 시스템 콜에는 주소 패밀리 식별자와 소켓 타입, 그리고 프로토콜을 인자로 준다. 먼저, 요구한 주소 패밀리를 사용하여 pops 벡터에서 일치하는 주소 패밀리가 있는지 찾는다. 어떤 주소 패밀리는 커널 모듈로 만들어져 있을 수도 있는데, 이 경우 kerneld 데몬이 이 모듈을 읽어들여야 작업을 계속할 수 있다. BSD 소켓을 나타내기 위해 새 socket 자료구조를 할당한다. 실질적으로 socket 자료구조는 물리적으로 VFS inode

자료구조의 한 부분이고 소켓을 할당한다는 것은 실제로는 VFS inode 를 할당한다는 것을 의미한다. 이는 소켓이 일반 파일과 똑같은 방법으로 작동한다는 것을 생각한다면 별로 이상하게 보이진 않을 것이다. 모든 파일은 VFS inode 자료구조로 나타내지며, 따라서 파일 함수들을 지원하려면 BSD 소켓 역시 VFS inode 자료구조로 표현되어야 한다.

새로 만들어진 BSD socket 자료구조는 주소 패밀리에 따라 특수한 소켓 루틴들에 대한 포인터를 가지고 있으며, 이는 pops 벡터에서 얻을 수 있는 proto_ops 자료구조에 설정된다. 타입은 요구한 소켓 타입으로 설정된다. 즉 SOCK_STREAM, SOCK_DGRAM 등등 중의 하나이다. 주소 패밀리에 따라 다른 생성 함수를 proto_ops 자료구조에 있는 주소를 이용하여 호출한다.

텅빈 파일 기술자(descriptor)가 현재 프로세스의 fd 벡터에서 할당되고, 이를 가리키는 file 자료구조가 초기화된다. 이는 파일 함수 포인터가 BSD 소켓 인터페이스에서 지원하는 BSD 소켓 파일 함수들을 가리키도록 설정하는 것을 포함한다. 이후의 작업들은 소켓 인터페이스로 전달되고 인터페이스는 차례로 주소 패밀리의 함수들을 호출함으로써 이들을 지원하는 주소 패밀리로 전달한다.

10.4.2 주소와 INET BSD 소켓을 바인드하기(binding)

들어오는 인터넷 접속 요구를 기다릴 수 (listen) 있으려면, 각 서버는 INET BSD 소켓을 만들어 이를 서버의 주소와 바인드해 주어야 한다. 이 바인드 작업은 대부분 INET 소켓계층이 아래 계층인 TCP 와 UDP 프로토콜 계층으로부터 어느 정도 지원을 받아 처리한다. 주소와 바인드 되어있는 소켓은 다른 통신을 위해서 사용할 수 없다. 이는 socket 의 상태는 TCP_CLOSE 여야만 한다는 것을 말한다. 바인드 함수에 전달된 sockaddr 은 바인드할 IP 주소와, 옵션으로 포트 번호를 가지고 있다. 보통은 INET 주소 패밀리를 지원하며 위에서 이 인터페이스를 사용할 수 있는 네트워크 장치에 할당된 IP 주소가, 여기서 바인드 되는 IP 주소이다. 현재 시스템에서 어떤 네트워크 인터페이스가 활성화되어 있는지는 ifconfig 명령을 사용하여 알 수 있다. IP 주소는 모두 1 이거나 모두 0 인 IP 브로드캐스트(broadcast) 주소일 수도 있다. 이들은 특별한 주소로서 "모든사람에게 보내라"를 의미한다. 또, 기계가 투명한 프록시나 방화벽으로 동작하고 있다면, 어떤 IP 주소하고도 바인드할 수 있다. 그러나 슈퍼유저 권한을 가진 프로세스만이 아무 IP 주소에나 바인드 할 수 있다. 바인드된 IP 주소는 recv_addr 에 있는 sock 자료구조와 saddr 항목에 저장된다. 이들은 해시로 찾을 때 쓰이며, 보내는 IP 주소로도 쓰인다. 포트 번호는 옵션이며 이를 지정하지 않으면 이를 지원하는 네트워크에게 아무것이나 비어있는 것을 달라고 요청한다. 관습적으로 1024 보다 작은 포트번호는 슈퍼유저 권한을 가지지 않은 프로세스는 사용할 수 없다. 만약 아래의 네트워크 계층에서 포트 번호를 할당한다면, 이는 항상 1024 보다 큰 것을 할당할 것이다.

아래기반의 네트워크 장치는 패킷을 받으면, 이를 올바른 INET 과 BSD 소켓으로 전달하여 처리될 수 있도록 해야 한다. 이런 이유로 UDP 와 TCP 는 들어온 IP 메시지에 있는 주소를 조회하여 올바른 socket/sock 쌍으로 전달하는데 사용할 수 있도록 해시 테이블을 관리한다. TCP 는 연결 지향 프로토콜이므로 UDP 패킷을 처리할 때보다 TCP 패킷을 처리하는데 더 많은 정보가 사용된다.

UDP 는 할당된 UDP 포트의 해시 테이블인 udp_hash 테이블을 관리한다. 이는 sock 자료구조의 포인터로서 포트 번호에 기반한 해시 함수로 인덱스되어 있다. UDP 해시 테이블은 허용되는 포트 번호의 수보다는 훨씬 적으므로 (udp_hash 는 128 또는 UDP_HTABLE_SIZE 의 값 만큼의 엔트리를 갖는다), 테이블의 어떤 엔트리들은 sock 자료구조의 연결 고리(이들은 sock 의 next 포인터로 서로 연결된다)를 가리킨다.

TCP 는 여러 개의 해시 테이블을 관리하므로 훨씬 더 복잡하다. 어쨌든 TCP 는 바인드 작업 동안에 바인드하는 sock 자료구조를 이의 해시 테이블에 실제로 추가하지는 않고, 단지 요구한 포트번호가 현재 사용되고 있는지만 검사한다. sock 자료구조는 리슨 작업을 하는 도중에 TCP 의 해시 테이블에 추가된다

REVIEW NOTE : 입력한 루트는 어떻게 되는가?

10.4.3 INET BSD 소켓으로 연결하기

소켓이 만들어지고, 이것이 내부로의 연결 요구를 받기 위한 용도로 사용되지 않았다면, 이는 외부로의 연결 요구에 사용할 수 있다. UDP 와 같은 비연결지향 프로토콜(connectionless protocol)에서는 이런 작업은 별로 하는 일이 없지만, TCP 같은 연결지향 프로토콜(connection oriented protocol)에서는 이는 두 개의 응용프로그램간에 가상 회로를 만드는 것을 포함한다.

외부로의 연결은 적절한 상태에 있는 INET BSD 소켓에서만 이루어질 수 있다 : 말하자면 이미 연결이 되어 있거나, 내부로의 연결을 기다리는데 사용하고 있는 것은 안된다는 것이다. 이는 BSD 소켓 자료구조가 SS_UNCONNECTED 상태에 있다는 것을 의미한다. UDP 프로토콜은 응용프로그램간에 가상 연결을 만들지 않는다. 보내는 메시지들은 모두 데이터그램이며, 메시지의 한 부분이 목적지에 도착할 수도, 도착하지 않을 수도 있다. 그렇긴 하지만, 접속 BSD 소켓 함수를 지원한다. UDP INET BSD 소켓에서의 접속 작업은 단순히 원격 응용프로그램의 주소 - IP 주소와 포트 번호 - 를 설정할 뿐이다. 추가적으로 라우팅 테이블 엔트리에 대한 캐시를 셋업하여, 이 BSD 소켓으로 보낸 UDP 패킷이 다시 라우팅 데이터베이스를 검사할 필요가 없도록 (이 루트가 틀린 것이 되기 전까지는) 한다. 캐시된 라우팅 정보는 INET `sock` 자료구조에서 `ip_route_cache` 가 가리키고 있다. 만약 아무런 주소 정보도 지정하지 않는다면, 이 캐시된 라우팅과 IP 주소 정보를 자동으로 BSD 소켓을 사용하여 보내는 메시지에 사용한다. UDP 는 `sock` 의 상태를 TCP_ESTABLISHED 로 바꾼다.

TCP BSD 소켓에서의 접속 작업에서는, TCP 는 접속 정보를 가진 TCP 메시지를 하나 만들어서 이를 주어진 IP 목적지로 보내야 한다. 이 TCP 메시지는 접속에 관련된 갖가지 정보들을 가지고 있다. 유일한 시작 메시지 순서 번호와 시작하는 (initiator) 호스트에서 처리할 수 있는 메시지의 최대 크기, 보내고 받는 윈도우 크기, 등등이 그것이다. TCP 에서는 모든 메시지에 번호가 붙으며, 초기 순서 번호는 첫번째 메시지 번호에 사용한다. 리눅스는 악의적인 프로토콜 공격을 피하기 위해 허용하는 범위 내에서 임의의 값을 고른다. 한쪽에서 전송한 메시지를 다른 쪽에서 성공적으로 받으면, 모든 메시지에 대해 그것이 성공적으로 깨지지 않고 도착했다는 것을 말하는 응답해 주어야 한다. 응답받지 않은 메시지는 다시 보내게 된다. 송수신 윈도우 크기는 응답을 보내지 않고 있을 수 있는 메시지의 수이다 (이만큼의 메시지를 보낼 때까지 ACK 가 오지 않아도 된다). 최대 메시지 크기는 요청을 시작한 쪽에서 사용하고 있는 네트워크 장치에 따른다. 만약 받는 쪽의 네트워크 장치가 이보다 작은 최대 메시지 크기를 지원한다면, 접속에서는 둘 중에 최소값을 사용하게 된다. 밖으로의 TCP 접속 요청을 하는 응용프로그램은 대상 응용프로그램이 이 접속 요구를 받거나 거부한다는 응답을 보낼 때까지 기다려야 한다. TCP `sock` 은 이제 메시지가 들어오길 기다려야 하므로, `tcp_listening_hash` 를 추가하여, 들어오는 TCP 메시지가 `sock` 자료구조로 갈 수 있게 한다. TCP 는 또한 대상 응용프로그램이 요구에 응답을 보내주지 않는 경우 밖으로의 접속 요구를 타임아웃 할 수 있도록 타이머를 시작한다.

10.4.4 INET BSD 소켓에서 리슨(listening)

소켓에 주소를 바인드 하였다면, 바인드한 주소를 지정하여 들어오는 접속 요구를 기다릴 수 있다. 네트워크 응용프로그램은 먼저 주소를 바인드 하지 않고도 접속을 기다릴 수 있는데, 이런 경우 INET 소켓 계층은 지금 프로토콜에서 사용하지 않고 있는 포트 번호를 찾아 이를 소켓에 자동으로 바인드 해준다. 리슨 소켓 함수는 소켓의 상태를 TCP_LISTEN 으로 바꾸고 들어오는 접속을 허가하는데 필요한 네트워크 특수 작업들을 한다.

UDP 소켓에 있어서는 소켓의 상태를 바꾸는 것으로도 충분하지만, TCP 는 소켓의 `sock` 자료구조를 두개의 해시 테이블에 추가하여 활성화되도록 한다. 이 두 개의 해시 테이블은 `tcp_bound_hash` 와 `tcp_listening_hash` 테이블이다. 둘 다 IP 포트 번호에 기반한 해

시 함수를 통하여 인덱스되어 있다.

활성화된 리슨 소켓에 대해 TCP 접속 요구가 들어오면, TCP 는 이를 나타내기 위해 새로운 sock 자료구조를 만든다. 이 sock 자료구조는 이 TCP 접속이 결국 받아들여진다면 TCP 접속의 하반부가 된다. 또한 접속 요구를 포함하고 있는 들어온 sk_buff 를 복사하여, 기다리는 sock 자료구조의 receive_queue 의 뒤에 이를 추가한다. 복사한 sk_buff 는 새로 만든 sock 자료구조에 대한 포인터를 갖는다.

10.4.5 접속 요구 허가하기(accepting)

UDP 는 접속이라는 개념을 지원하지 않으므로, INET 소켓 접속을 허락하는 것은 TCP 프로토콜에만 적용이 되며, 접속을 기다리는 소켓에서 접속을 허락하는 것은 원래의 기다리는 소켓에서 socket 자료구조를 복사하여 새로운 socket 을 만든다. 허가 작업은 자신을 지원하는 프로토콜 계층, 이 경우 INET 계층으로 넘어가서 들어오는 어떤 접속 요구를 받아들이라고 한다. 만약 아래 계층의 프로토콜이 UDP 같이 접속을 지원하지 않는 것이라면 이 접속 허가 과정은 실패한다. 그렇지 않으면 접속 허가 과정은 실제 프로토콜, 이 경우 TCP 로 전달된다. 이 접속 허가 작업은 블럭킹 모드일수도, 블럭킹 모드가 아닐수도 있다. 블럭킹 모드가 아닌 경우, 만약 아무런 들어오는 접속이 없으면, 이 접속 작업은 실패하고, 새로 만들어진 socket 자료구조는 버려질 것이다. 블럭킹 모드인 경우, 접속 허가를 하는 네트워크 응용프로그램은 대기 큐에 들어가고 TCP 접속 요구를 받을 때까지 중단된다. 접속 요구가 들어오면, 그 요구를 갖고 있는 sk_buff 는 무시되고, sock 자료구조는 이전에 만든 새 socket 자료구조와 연결되어 있는 INET 소켓 계층으로 되돌아간다. 네트워크 응용프로그램에 새로 만들어진 소켓의 파일 기술자(fd)를 돌려주고, 응용 프로그램은 새로 만들어진 BSD 소켓을 가지고 소켓 작업을 하는데 이 파일 기술자를 사용할 수 있다.

10.5 IP 계층

10.5.1 소켓 버퍼(Socket Buffer)

많은 네트워크 프로토콜 계층을 가지고, 각각이 다른 것의 서비스를 사용하는 방법의 문제중의 하나는, 각 프로토콜이 전송하는 데이터에 프로토콜 헤더와 꼬리를 붙이고, 받은 데이터를 처리할 때 이를 제거해야 한다는 것이다. 이는 각 프로토콜 계층마다 특별한 프로토콜 헤더와 꼬리를 찾아야 하므로 프로토콜 사이에 데이터 버퍼를 전달하는 것을 어렵게 만든다. 방법중의 하나는 각 계층마다 버퍼를 복사하는 것이지만, 이는 매우 비효율적이다. 대신 리눅스는 프로토콜 계층 사이와 네트워크 디바이스 드라이버 간에 데이터를 전달하기 위해 sk_buffs 라는 소켓 버퍼를 사용한다. sk_buffs 는 포인터와 길이 항목을 가지고 있어서 각 프로토콜 계층이 표준 함수를 통해 응용프로그램 데이터를 다룰 수 있게 한다.

include/linux/skbuff.h 참조

그림 10.4 는 sk_buff 자료구조를 보여준다. 각 sk_buff 는 자신과 연관된 데이터 블럭을 가지고 있다. sk_buff 는 네개의 데이터 포인터를 가지고 있는데, 이들은 소켓 버퍼 데이터를 다루고 관리하는데 사용된다.

헤드(head) 메모리에서 데이터의 시작을 가리킨다. 이는 sk_buff 와 이와 관련된 데이터 블럭을 할당할 때 고정된다.

데이터(data) 현재 프로토콜 데이터의 시작을 가리킨다. 이 포인터는 현재 sk_buff 를 소유하고 있는 프로토콜 계층에 따라 달라진다.

꼬리(tail) 현재 프로토콜 데이터의 끝을 가리킨다. 마찬가지로, 소유하고 있는 프로토콜 계층에 따라 달라진다.

끝(end) 메모리에서 데이터 영역의 끝을 가리킨다. sk_buff 를 할당할 때 결정된다.

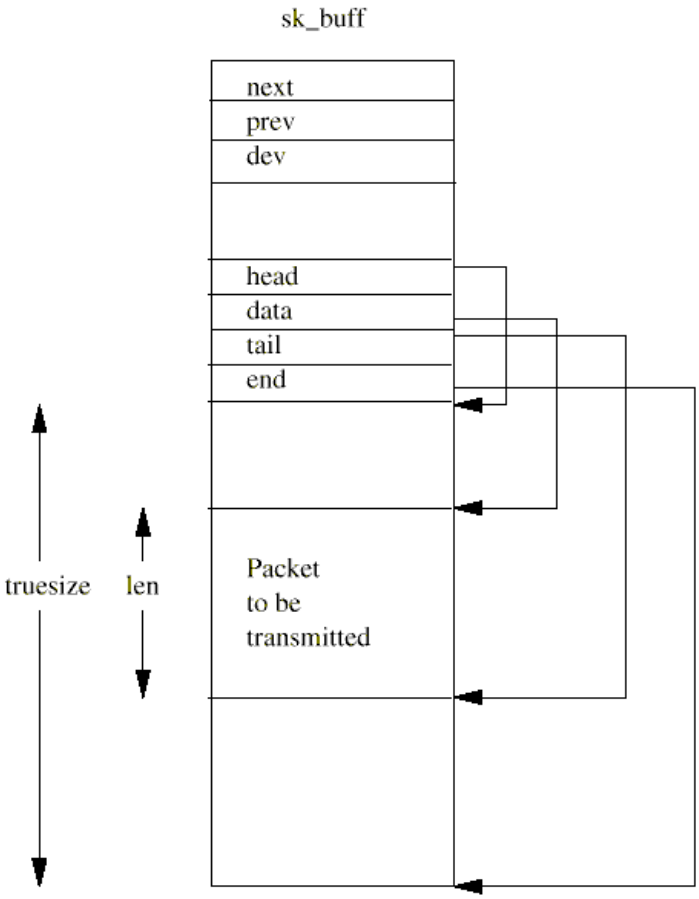


그림 10.4 : 소켓 버퍼 (sk_buff)

길이를 나타내는 항목으로는 `len` 과 `truesize` 두개가 있으며, 이들은 각각 현재 프로토콜 패킷의 길이와, 상대적인 데이터 버퍼의 전체 크기를 나타낸다. `sk_buff` 를 다루는 코드는 응용프로그램 데이터에 프로토콜 헤더와 꼬리를 붙이고 제거하는 표준적인 방법들을 제공한다. 이들은 안전하게 `sk_buff` 에 있는 `data`, `tail`, 그리고 `len` 항목들을 다룬다.

include/linux/skbuff.h skb_push() 참조

push `data` 포인터를 데이터 영역의 시작쪽으로 이동하고, `len` 항목을 증가시킨다. 이는 전송할 데이터의 시작부분에 데이터나 프로토콜 헤더를 붙이는데 사용된다.

include/linux/skbuff.h skb_pull() 참조

pull `data` 포인터를 시작부분에서 먼 쪽으로, 데이터 영역의 끝쪽으로 이동하고, `len` 항목을 감소시킨다. 이는 수신한 데이터의 시작부분에서 데이터나 프로토콜 헤더를 제거하는데 사용된다.

include/linux/skbuff.h skb_put() 참조

put `tail` 포인터를 데이터 영역의 끝쪽으로 이동하고 `len` 항목을 증가시킨다. 이는 전송할 데이터의 끝에 데이터나 프로토콜 정보를 추가하는데 사용된다.

include/linux/skbuff.h skb_trim() 참조

trim `tail` 포인터를 데이터 영역의 시작쪽으로 이동하고 `len` 항목을 감소시킨다. 이는 수신한 패킷에서 데이터나 프로토콜 꼬리를 제거하는데 사용된다.

`sk_buff` 자료구조는 또한 처리도중에 `sk_buff` 의 이중 원형 연결 리스트에 저장하는데 사용하는 포인터들을 가지고 있다. 그리고 `sk_buffs` 를 이들 리스트의 앞이나 뒤에 추가하고 제거하는데 사용하는 일반적인 `sk_buff` 루틴들도 있다.

10.5.2 IP 패킷 수신하기

커널에서 리눅스 드라이버들이 어떻게 만들어지고 초기화되는지는 8 장에서 설명했다. 이 초기화의 결과는 dev_base 리스트에서 서로 연결되어 있는 일련의 device 자료구조이다. 각 device 자료구조는 장치를 서술하고, 네트워크 프로토콜 계층에서 네트워크 드라이버가 어떤 일을 수행해야 할 때 부를 수 있는 콜백 루틴 세트를 제공한다. 이들 함수들은 대부분 데이터 전송과 네트워크 장치의 주소에 관련되어 있다. 네트워크 장치가 네트워크로부터 패킷을 수신하면 이 수신한 데이터를 sk_buff 자료구조로 바꾸어야 한다. 네트워크 드라이버는 이들을 수신할 때마다 backlog 큐에 수신한 sk_buff 들을 추가한다. 만약 backlog 큐가 너무 커지면, 수신한 sk_buff 들은 무시된다. 이제 해야 할 일이 있으므로 실행할 준비가 되었다고 네트워크 하반부(bottom half)에 표시한다.

net/core/dev.c
netif_fx() 참조

스케줄러가 네트워크 하반부 핸들러를 실행하면, 이는 sk_buff 의 backlog 큐를 처리하기 이전에 수신한 패킷을 어떤 프로토콜 계층으로 전달할지를 결정하며 전송되길 기다리고 있는 네트워크 패킷들을 처리한다. 리눅스 네트워킹 계층을 초기화할 때 각 프로토콜은 packet_type 자료구조를 ptype_all 리스트나 ptype_base 해시테이블에 추가함으로서 자신들을 등록했다. packet_type 자료구조는 프로토콜 타입과 네트워크 장치에 대한 포인터, 프로토콜의 수신 데이터 처리 루틴, 그리고 마지막으로 리스트나 해시 고리에 있는 다음 packet_type 자료구조에 대한 포인터를 가지고 있다. ptype_all 고리는 어떤 네트워크 장치이든지부터 수신되는 모든 패킷들을 엿보는데(snoop) 사용되지만 잘 사용되지 않는다. ptype_base 해시 테이블은 프로토콜 식별자로 해시되어 있으며, 들어오는 네트워크 패킷을 어떤 프로토콜이 받을 것인지 결정하는데 사용된다. 네트워크 하반부는 들어오는 sk_buff 의 프로토콜 타입과 각 테이블에 있는 하나 이상의 packet_type 엔트리와 매치시킨다. 프로토콜은 하나 이상의 엔트리와 매치될 수 있는데, 예를 들어 모든 네트워크 트래픽을 엿볼 때 같은 경우이며, 이 경우 sk_buff 는 복제가 된다. sk_buff 는 매치되는 프로토콜 처리 루틴으로 전달된다.

net/core/dev.c
net_bh() 참조

net/ipv4/ip_input.c
ip_recv() 참조

10.5.3 IP 패킷 전송하기

패킷은 응용프로그램이 데이터를 교환하거나, 네트워크 프로토콜이 이미 만들어진 연결이나 만들어지는 연결을 지원할 때 만들어져서 보내진다. 어떤 방법으로 데이터가 만들어졌던지 간에 데이터를 포함하고 있는 sk_buff 가 만들어지고, 각 프로토콜 계층을 통과하면서 프로토콜 계층이 다양한 헤더를 붙인다.

sk_buff 는 전송할 네트워크 장치로 전달되어야 한다. 먼저 IP 같은 프로토콜이라도 어떤 네트워크 장치를 사용할지를 결정해야 한다. 이는 패킷에 가장 맞는 루트에 따라 다르다. PPP 프로토콜같은 것을 통해 모뎀으로 하나의 네트워크에 연결된 컴퓨터에 있어서는 이 루트를 선택하는 것은 쉽다. 패킷은 루프백 장치를 통해 로컬호스트나, PPP 모뎀 연결의 끝에 있는 게이트웨이 둘 중 하나로 전송될 것이다. 이더넷으로 연결되어 있는 컴퓨터에 있어서는, 네트워크에 많은 컴퓨터가 연결되어 있으므로 이 선택은 더 어렵다.

include/net/
route.h 참조

IP 패킷을 전송할 때 항상 IP 는 도달할 IP 주소로 가는 루트(route)를 해결하기 위해 라우팅 테이블(routing table)을 사용한다. 각 IP 목적지는 라우팅 테이블에서 성공적으로 찾게 되어, 사용할 루트를 기술하는 rtable 자료구조를 돌려준다. 이는 사용할 출발지 IP 주소와, 네트워크 device 자료구조의 주소, 때때로 미리 만들어진 하드웨어 헤더를 포함한다. 이 하드웨어 헤더는 네트워크 장치마다 다른 것으로서 출발지와 도착지의 하드웨어 주소와, 매개체별로 다른 정보를 가지고 있다. 만약 네트워크 장치가 이더넷 장치이라면, 하드웨어 헤더는 그림 10.1 에서 보는 바와 같을 것이며, 출발지와 도착지 주소는 물리적인 이더넷 주소일 것이다. 하드웨어 헤더는 루트와 함께 캐시되는데, 이는 이 하드웨어 헤더가 이 루트를 통하여 전송하는 모든 IP 패킷에 추가되어야 하는데, 이를 다시 만드는 것은 시간이 걸리기 때문이다. 하드웨어 헤더는 ARP 프로토콜로 해결되어야 하는 물리적인 주소를 가질 수도 있다. 이 경우 밖으로 나가는 패킷은 주소가 해결될 때까지 꼼짝못하고 기다리고 있어야 한다. 한번 주소가 해결되고 나면, 하드웨어 헤더가 만들어지고, 이 인터페이스를 사용하는 IP 패킷이

다시 ARP 를 할 필요가 없도록 이 하드웨어 헤더를 캐시한다.

10.5.4 데이터 조각내기 (data fragmentation)

모든 네트워크 장치는 최대 패킷 크기를 가지고 있으며, 이보다 큰 크기의 데이터를 보내거나 받을 수 없다. IP 프로토콜은 이런 경우를 허용하여 데이터를 네트워크 장치가 처리할 수 있는 패킷 크기로 데이터를 잘게 쪼갠다. IP 프로토콜 헤더는 플래그와 이 조각의 옵셋을 담은 조각 항목을 가지고 있다.

net/ipv4/ip_input.c ip_build_xmit() 참조

IP 패킷이 전송할 준비가 되면, IP 는 IP 패킷을 밖으로 보낼 네트워크 장치를 찾는다. 장치는 IP 라우팅 테이블에서 찾게 된다. 각 `device` 는 최대 전송 단위를 나타내는 항목으로 가지고 있는데 (바이트 단위), 이는 `mtu` 항목이다. 만약 장치의 `mtu` 가 전송하려는 IP 패킷의 크기보다 작으면, IP 패킷은 좀 더 작은 크기(`mtu` 크기)의 조각으로 쪼개져야 한다. 각 조각은 `sk_buff` 로 표현된다. IP 헤더에는 이것이 조각이며, 이 패킷이 데이터의 어떤 옵셋부터 가지고 있는지 표시된다. 마지막 패킷은 마지막 IP 조각이라고 표시된다. 만약, 이 쪼개는 도중에 IP 가 `sk_buff` 를 할당받지 못한다면 전송을 실패하게 된다.

net/ipv4/ip_input.c ip_rcv() 참조

IP 조각을 수신하는 것은 전송하는 것보다 더 어려운데, 이는 IP 조각이 아무런 순서로나 도착할 수 있으므로 모두 수신받아야 재조립할 수 있기 때문이다. IP 패킷을 수신할 때마다 이것이 IP 조각인지 검사한다. 메시지 조각이 처음 도착하면, IP 는 새 `ipq` 자료구조를 만들고, 이를 재조립을 기다리는 IP 조각의 리스트인 `ipqueue` 에 연결한다. IP 조각이 계속 수신되면 맞는 `ipq` 자료구조를 찾아 이 조각을 나타낼 `ipfrag` 자료구조를 새로 만든다. 각 `ipq` 자료구조는 조각난 IP 수신 프레임을 출발지와 도착지 IP 주소와 함께 유일하게 기술하며, 위 계층 프로토콜 식별자와 이 IP 프레임의 식별자를 기술한다. 모든 조각이 도착하면, 이들은 하나의 `sk_buff` 로 합쳐지고 처리할 다음 프로토콜 계층으로 전달된다. 각 `ipq` 는 제대로 된 조각이 도착할 때마다 다시 시작되는 타이머를 가지고 있다. 만약 이 타이머가 만료되면, `ipq` 자료구조와 이것의 `ipfrag` 들은 소멸되며, 메시지는 전송 중에 사라진 것으로 간주된다. 이 메시지를 다시 전송하는 것은 더 윗 레벨의 프로토콜이 담당하는 문제이다.

10.6 주소 결정 프로토콜(Address Resolution Protocol, ARP)

net/ipv4/ip_input.c ip_build_xmit() 참조

주소 결정 프로토콜의 역할은 IP 주소에서 이더넷 주소와 같은 물리적 하드웨어 주소로의 변환을 제공하는 것이다. IP 는 데이터를 전송할 디바이스 드라이버에게 전달하기 (`sk_buff` 의 형태로) 바로 전에 이런 변환을 필요로 한다. 이는 이 장치가 하드웨어 헤더를 필요로 하는지, 만약 그렇다면 이 패킷용으로 하드웨어 헤더를 다시 만들어야 하는지 알기 위해 여러가지 검사를 수행한다. 리눅스는 하드웨어 헤더를 자주 다시 만들지 않도록 이를 캐시한다. 만약 하드웨어 헤더를 다시 만들 필요가 있다면, 장치 고유의 하드웨어 헤더 재제작 루틴을 호출한다. 모든 이더넷 장치는 똑같은 일반적인 헤더 재제작 루틴을 사용하며, 이 루틴은 목적지 IP 주소를 물리적인 주소로 바꾸기 위해 차례로 ARP 서비스를 사용한다.

net/ethernet/eth.c eth_rebuild_ header() 참조

ARP 프로토콜 그 자체는 매우 단순하며, ARP 요구와 ARP 응답 두가지 메시지 형태로 이루어져 있다. ARP 요구는 변환을 필요로 하는 IP 주소를 가지고 있고, 응답은 (바라건데) 변환된 IP 주소인 하드웨어 주소를 가지고 있다. ARP 요구는 네트워크에 연결된 모든 호스트로 방송(브로드캐스트) 되므로, 이더넷 네트워크에서는 이더넷에 연결된 모든 기계들이 이 ARP 요구를 받게 된다. 이 요구에 있는 IP 주소를 소유하고 있는 기계는 이 ARP 요구에 응답하여 자신의 물리적인 주소를 담고 있는 ARP 응답으로 답하게 된다.

리눅스에서 ARP 프로토콜 계층은 각각 IP 에서 물리주소로의 변환을 나타내는 `arp_table` 자료구조의 테이블을 가지고 이루어져 있다. 이들 엔트리들은 IP 주소가 변환될 필요가 있을 때 만들어지고, 시간이 지나 낡아지면 제거된다. 각 `arp_table` 자료구조는 다음과 같은 항목들을 가진다 :

마지막 사용(last used)	ARP 엔트리가 마지막으로 사용된 시간
마지막 갱신(last updated)	ARP 엔트리가 마지막으로 갱신된 시간
플래그(flags)	엔트리가 완료되었는지 같은 엔트리의 상태를 나타낸다.
IP 주소	엔트리가 나타내는 IP 주소
하드웨어 주소	변환된 하드웨어 주소
하드웨어 헤더	캐시된 하드웨어 헤더에 대한 포인터
타이머(timer)	응답하지 않는 ARP 요구를 타임아웃 시키는데 사용하는 timer_list 엔트리
재시도(retries)	이 ARP 요구를 재시도한 횟수
sk_buff 큐	이 IP 주소를 해결하기 기다리는 sk_buff 엔트리의 리스트

ARP 테이블은 arp_table 엔트리들을 잇기 위해 포인터의 테이블로 되어 있다 (arp_tables 벡터). 엔트리들은 이들에 대한 접근 속도를 높이기 위해 캐시되며, 각 엔트리는 IP 주소의 끝 두 바이트를 가져와 테이블에 대한 인덱스를 계산하고, 원하는 것을 찾을 때까지 해시 테이블에서 엔트리의 고리를 따라가 찾게 된다. 리눅스는 또한 미리 만들어진 하드웨어 헤더를 hh_cache 자료구조 형태로 arp_table 엔트리에 캐시시킨다.

IP 주소변환을 요구했는데 일치하는 arp_table 엔트리가 없을 경우, ARP 는 ARP 요구 메시지를 보내야 한다. ARP 는 arp_table 에서 새 arp_table 엔트리를 만들고, 주소 변환을 필요로 하는 패킷들을 포함하고 있는 sk_buff 를 새로 만들어진 엔트리의 sk_buff 큐에 큐시킨다. ARP 는 ARP 요구를 보내고 ARP 만료 타이머를 실행한다. 아무런 응답이 없다면 ARP 는 여러번 재시도를 하고, 여전히 응답이 없다면 ARP 는 arp_table 엔트리를 제거한다. IP 주소가 변환되기를 기다려 큐되어 있는 어떤 sk_buff 자료구조이든 간에 통지를 받게 되고, 이런 실패와 협조하는 것은 이들을 전송하려는 프로토콜 계층의 몫이다. UDP 는 잃어버린 패킷에 대해서 신경을 쓰지 않지만, TCP 는 성립된 TCP 링크를 통하여 재전송하려고 시도할 것이다. 만약 IP 주소의 소유자가 하드웨어 주소를 돌려주며 응답한다면, arp_table 엔트리는 완료된 것으로 표시되고, 큐되어 있는 모든 sk_buff 들은 큐에서 재거되고 전송될 것이다. 하드웨어 주소는 각 sk_buff 의 하드웨어 헤더에 기록된다.

ARP 프로토콜 계층은 자신의 IP 주소를 지정하고 있는 ARP 요구에 반드시 응답해야 한다. 이 계층은 자신의 프로토콜 타입 (ETH_P_ARP)를 등록하고, packet_type 자료구조를 생성한다. 이는 네트워크 장치가 수신한 모든 ARP 패킷을 전달받게 된다는 것을 의미한다. 이는 ARP 응답뿐만 아니라 ARP 요구도 포함한다. 이는 수신한 장치의 device 자료구조에 저장되어 있는 하드웨어 주소를 사용하여 ARP 응답을 만든다.

네트워크 구성은 시간이 지나면서 변할 수 있으며, IP 주소는 다른 하드웨어 주소로 다시 할당될 수도 있다. 예를 들어, 어떤 전화접속 서비스는 연결이 될 때마다 각각 다른 IP 주소를 배정한다. ARP 테이블이 가장 최근의 엔트리를 가질 수 있도록, ARP 는 정기적인 타이머를 돌려서 모든 arp_table 엔트리들이 타임아웃이 되지 않았는지 살펴본다. 이는 하나 이상의 캐시된 하드웨어 헤더를 갖고 있는 엔트리들을 제거하지 않도록 매우 조심한다. 이들 엔트리를 지우는 것은 다른 자료구조들이 이에 의존하고 있으므로 매우 위험하다. 어떤 arp_table 엔트리들은 영구적이며, 이들은 할당이 해제되지 않도록 표시가 된다. ARP 테이블은 너무 커지면 안된다. 각 arp_table 엔트리는 어느정도 커널 메모리를 잡아먹기 때문이다. 새 엔트리가 할당되어야 하고 ARP 테이블이 최대 크기에 도달할 때마다, 테이블은 가장 오래된 엔트리들을 찾아 이를 제거한다.

10.7 IP 라우팅(routing)

IP 라우팅 함수는 특정 IP 주소를 목적지로 가진 IP 패킷을 어디로 보낼지를 결정한다. IP 패킷을 전송할 때 많은 선택을 할 수 있다. 목적지에 결국 도착할 수 있을까? 만약 도착할 수 있다면, 전송하는데 어떤 네트워크 장치를 사용할 것인가? 목적지에 도착하는데 사용할 수 있는 네트워크 장치가 하나 이상 있다면, 어떤 것이 더 좋은 것인가? IP 라우팅 데이터베이스는 이들 질문에 대답할 수 있는 정보를 관리한다. 여기에 두가지 데이터베이스가 있는

데, 가장 중요한 것은 전달 정보 데이터베이스(Forwarding Information Database)이다. 이것은 IP 주소와 가장 좋은 길에 대해서 알려진 것들의 소모적인 목록이다. IP 목적지로의 길을 빨리 찾기 위해, 더 작고 더 빠른 데이터베이스인 루트 캐시(route cache)가 사용된다. 다른 모든 캐시처럼 이는 자주 접근하는 길들에 대해서만 가지고 있어야 한다; 이것의 내용은 전달 정보 데이터베이스에서 가져온 것이다.

루트는 BSD 소켓 인터페이스로 IOCTL 요구를 보냄으로써 추가되거나 삭제된다. 이들은 프로토콜에서 프로세스로 전달된다. INET 프로토콜 계층은 IP 루트를 추가하거나 삭제하는데 슈퍼유저 권환을 가진 프로세스만을 허가한다. 이들 루트들은 고정될 수도 있고, 시간이 지나면서 동적으로 변할 수도 있다. 대부분의 시스템은 라우터가 아니라면 고정된 루트를 사용한다. 라우터는 지속적으로 모든 알려진 IP 목적지로 가는 길들의 유효성을 검사하는 라우팅 프로토콜을 실행한다. 라우터가 아닌 시스템들은 단말 시스템이라고 한다. 라우팅 프로토콜은 GATED 같은 데몬으로 구현되어 있으며, 마찬가지로 IOCTL BSD 소켓 인터페이스를 통하여 루트를 추가하거나 삭제한다.

10.7.1 루트 캐시(Route Cache)

IP 루트를 조회하면 일치하는 루트를 찾기 위해 루트 캐시를 먼저 검사한다. 루트 캐시에 일치하는 루트가 없다면 전달 정보 데이터베이스에서 루트를 찾는다. 만약 아무런 루트도 찾을 수 없다면, IP 패킷은 전송에 실패하고 이를 응용프로그램에 알린다. 만약 루트가 전달 정보 데이터베이스에 있고 루트 캐시에 없다면, 이 루트에 해당하는 새 엔트리를 만들어 루트 캐시에 추가한다. 루트 캐시는 `rtable` 자료구조의 연결고리에 대한 포인터를 가지고 있는 테이블(`ip_rt_hash_table`)이다. 루트 테이블에서의 인덱스는 IP 주소의 하단 두 바이트에 기반한 해시함수이다. 이들 두 바이트는 목적지마다 가장 달라서 해시값을 가장 잘 분산시켜 줄 수 있는 것이다. 각 `rtable` 엔트리는 루트에 대한 정보 - 목적 IP 주소와 이 IP 주소에 도달하는데 사용할 네트워크 `device`, 사용할 수 있는 메시지의 최대 크기 등등 - 를 가지고 있다. 이는 또한 참조횟수도 가지고 있는데, 이는 사용횟수와 이것이 사용된 마지막 시간의 타임스탬프를 가지고 있다 (`jiffies` 값으로). 참조횟수는 이 루트가 사용될 때마다 증가하여, 이 루트를 사용하는 네트워크 연결의 숫자를 보여준다. 이는 응용프로그램이 이 루트를 사용하기를 그만두면 감소한다. 사용횟수는 이 루트를 찾았을 때마다 증가하며, `rtable` 해시 고리에서 이 엔트리의 순서를 결정하는데 사용된다. 루트 캐시에 있는 모든 엔트리에 있는 마지막 사용한 타임스탬프를 정기적으로 검사하여 `rtable` 이 너무 오래되지 않았는지 살핀다. 만약 루트가 최근에 사용되지 않았다면 루트 캐시에서 빠지게 된다. 만약 루트가 루트 캐시에 있다면, 이 루트는 가장 많이 사용한 엔트리가 해시 고리의 맨 앞에 오도록 배치된다. 이는 루트를 조회할 때 빨리 찾게 된다는 것을 의미한다.

net/ipv4/route.c
ip_rt_check_
expire() 참조

10.7.2 전달 정보 데이터베이스(Forwarding Information Database)

전달 정보 데이터베이스(그림 10.5 에서 보여주고 있다)는 어떤 시간에 시스템에서 사용할 수 있는 루트들을 IP 관점에서 가지고 있다. 이는 매우 복잡한 자료구조이며, 상당히 효과적으로 배치되어 있지만, 참고하기에 빠른 데이터베이스는 아니다. 특히, 전송하는 모든 IP 패킷마다 목적지를 이 데이터베이스에서 찾게 된다면 매우 느릴 것이다. 이는 루트 캐시가 있는 이유이기도 하다 - 알고 있는 좋은 루트를 사용하여 IP 패킷 전송하는 것을 더 빠르게 하기. 루트 캐시는 전달 정보 데이터베이스에서 파생된 것으로 자주 사용하는 엔트리들을 대표한다.

각 IP 서브넷은 `fib_zone` 자료구조로 표현한다. 이들 모두는 `fib_zone`s 해시 테이블에서 가리키고 있다. 해시 인덱스는 IP 서브넷 매스크에서 만들어진다. 똑같은 서브넷으로의 모든 루트들은 `fib_node` 의 쌍으로 나타내지며, `fib_info` 자료구조는 각 `fib_zone` 자료구조의 `fz_list` 로 큐된다. 만약 이 서브넷에 있는 루트의 숫자가 커지면, `fib_node` 자료구조를 쉽게 찾기 위해 해시테이블이 만들어진다.

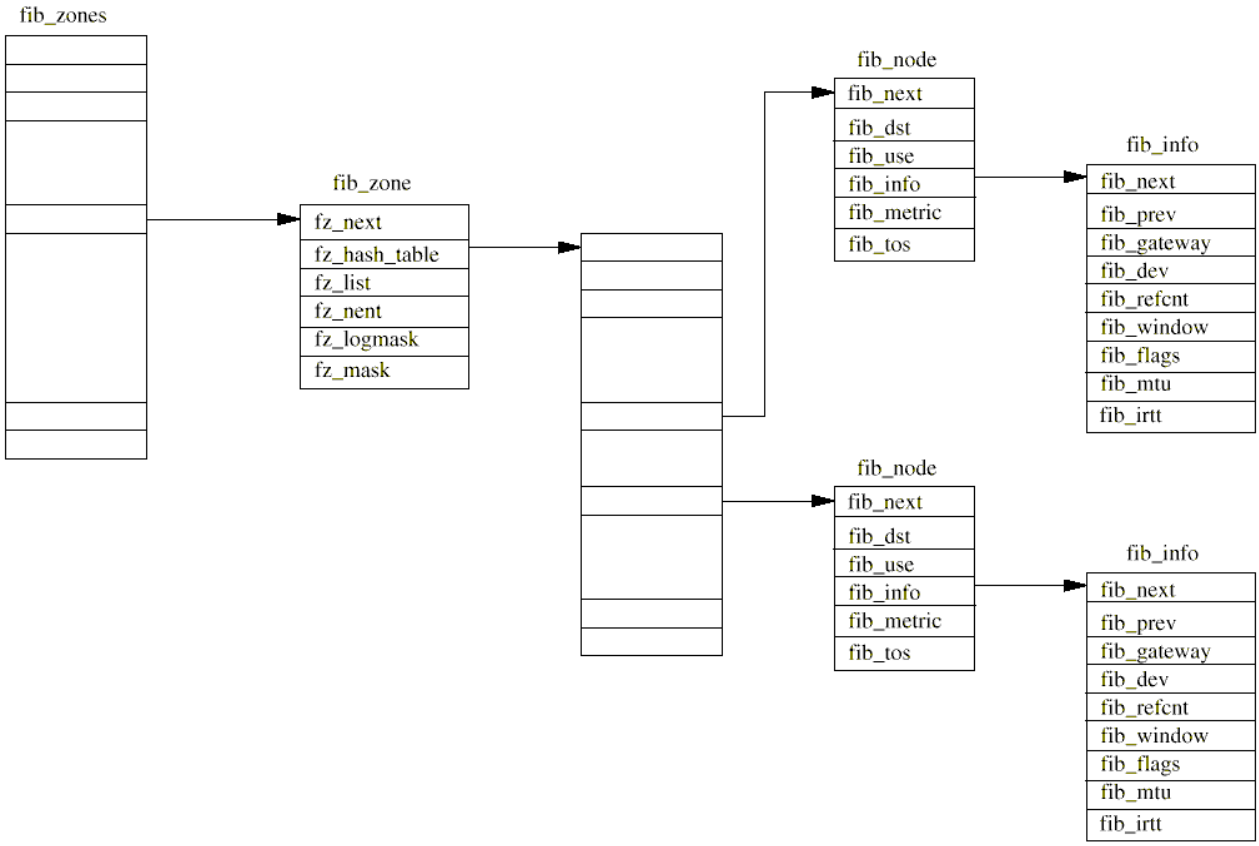


그림 10.5 : 전달 정보 데이터베이스

똑같은 IP 서브넷에 여러개의 루트가 있을 수 있으며, 이들 루트들은 여러 게이트웨이 중의 하나를 통하게 된다. IP 라우팅 계층은 똑같은 게이트웨이를 사용하여 하나의 서브넷으로 여러 개의 루트가 있는 것을 허가하지 않는다. 다르게 말하면, 서브넷으로 가는 루트가 여러 개가 있다면, 각 루트는 다른 게이트웨이를 사용하도록 하여야 한다는 것이다. 각 루트와 연관되어 있는 것은 그것의 거리(metric)이다. 이것은 이 경로가 얼마나 유리한지를 측정하게 하는 것이다. 한 루트의 거리는 본질적으로 목적하는 서브넷에 도착하기까지 거쳐야 하는 IP 서브넷의 수이다. 이 값이 더 클 수록 더 좋지 않은 루트이다.

번역 : 김성룡, 이호, 홍경선
정리 : 심마로, 이호

11 장

커널 메커니즘 (Kernel Mechanism)

이 장에서는 커널의 여러 부분들이 함께 효과적으로 동작할 수 있도록 리눅스 커널이 제공하는 몇가지 일반적인 작업과 메커니즘에 대해서 설명한다.

11.1 하반부 처리(Bottom Half Handling)

커널에서는 종종 꼭 그 시점에서 일을 처리하길 바라지 않는 경우가 있다. 이의 대표적인 예로 인터럽트를 처리하는 도중이다. 인터럽트가 발생했을 때 프로세서는 자신이 하던 일을 중지하고 운영체제는 인터럽트를 해당하는 디바이스 드라이버에게 전달한다. 인터럽트를 처리하는 동안에는 시스템의 다른 부분을 실행할 수 없으므로, 디바이스 드라이버는 인터럽트 처리에 너무 많은 시간을 보내면 안된다. 여기에는 당장이 아니라 나중에 처리해도 되는 일들이 종종 있다. 리눅스의 하반부 핸들러(bottom half handler)[105]는 디바이스 드라이버나 리눅스 커널의 다른 부분들이, 할 일을 나중에 실행되는 큐에 넣을 수 있도록 하기 위해 개발되었다. 그림 11.1 은 하반부 처리와 관련된 커널의 자료구조를 보여준다. 모두 32 개까지의 서로 다른 하반부 핸들러가 있을 수 있다[106]. `bh_base` 는 커널의 하반부 핸들러 루틴을 가리키고 있는 포인터들의 벡터이다. `bh_mask` 와 `bh_active` 는 어떤 핸들러가 설치되어 있고 액티브 한 지 나타내는 비트들의 집합이다. `bh_mask` 의 비트 N 이 설정되어 있다면 `bh_base` 의 N 번째에 하반부 루틴이 담겨 있는 것이다. `bh_active` 의 N 번째 비트가 설정되어 있으면, 스케쥴러가 가능하다고 판단할 때 N 번째 하반부 핸들러 루틴을 되도록 빨리 불러주어야 한다는 것이다. 이들 인덱스들은 정적으로 정의된 것이다[107]. 타이머 하반부 핸들러는 가장 높은 우선순위를 가지며(인덱스 0), 콘솔 하반부 핸들러는 다음 우선순위(인덱스 1)를 가진다. 일반적으로 하반부 핸들러 루틴들은 자신과 연결된 작업들의 목록을 가지고 있다. 예를 들어, 즉시실행(immediate) 하반부 핸들러는 바로 수행해야 하는 작업들의 목록인 즉시실행 작업 큐(`tq_immediate`)를 가지고 동작한다.

include/linux/interrupt.h 참조

커널의 하반부 핸들러 중에 어떤 것들은 장치에 고정되어 있지만 다른 것들은 보다 일반적

역주 [105]) 하반부(bottom half)라는 말은 인터럽트 핸들러를 상반부(top half)라고 생각하여 인터럽트 핸들러에서 처리되지 않고 나중으로 미뤄진 작업을 대비시켜 붙인 이름이다. (flyduck)

역주 [106]) 이는 하반부 처리에 관련된 자료구조가 4바이트 크기의 마스크와 고정된 크기의 배열로 되어 있기 때문이다. 따라서 하반부 처리를 사용할 수 있는 것은 한정되어 있으며, 이것보다 좀 더 개선된 구조의 작업큐가 나오게 된다. (flyduck)

역주 [107]) 정적(static)으로 정의되었다는 의미는, 하반부 핸들러를 사용하겠다고 동적으로 인덱스를 얻어서 사용하는 것이 아니라, 미리 각 인덱스에는 무엇이 담길 것이며 이 인덱스를 정의하는 상수(아래에 나오는)가 정의되어 있다는 것이다. (flyduck)

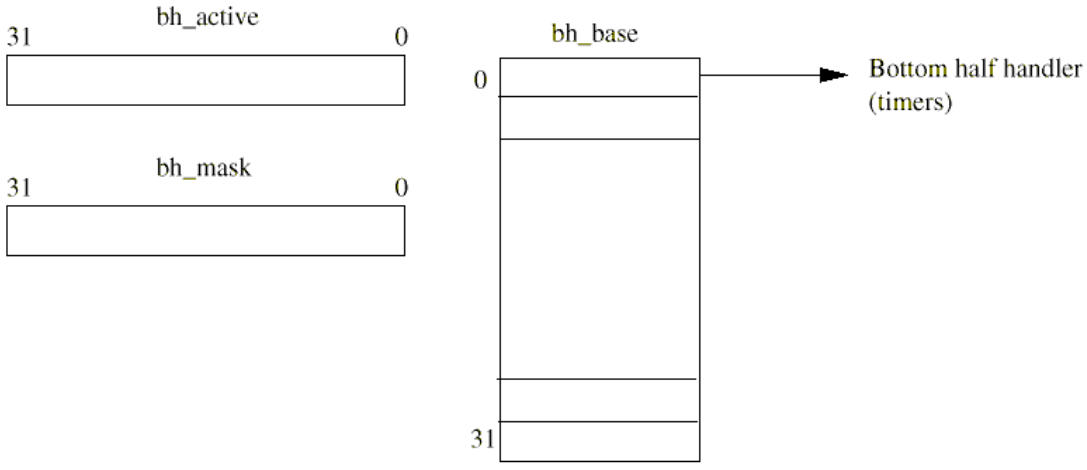


그림 11.1 : 하반부 처리 자료구조

으로 쓸 수 있다[108] :

TIMER 이 핸들러는 시스템의 주기적으로 발생하는 타이머 인터럽트가 발생할 때마다 액티브로 표시가 되고, 커널의 타이머 큐 메커니즘을 위해 사용된다.

CONSOLE 이 핸들러는 콘솔 메시지를 처리하는데 사용된다.

TQUEUE 이 핸들러는 `tty` 메시지를 처리하는데 사용된다[109].

NET 이 핸들러는 일반적인 네트웍 처리에 사용된다.

IMMEDIATE 이는 여러 디바이스 드라이버들이 나중에 실행될 작업들을 쌓아두는데 사용된다.

kernel/softirq.h
do_bottom_half()
참조

디바이스 드라이버나 커널의 어떤 부분이 나중에 수행될 작업을 스케쥴할 필요가 있을 때, 이들은 작업을 적당한 시스템 큐에 - 예를 들어 타이머 큐같은 - 넣고, 커널에 하반부 핸들러가 수행될 필요가 있다고 신호를 보낸다. 이는 `bh_active` 의 해당하는 비트를 설정하게 된다[110]. 만약 드라이버가 어떤 일을 즉시실행 큐에 넣고 이 즉시실행 하반부 핸들러가 실행되어 이를 처리하길 바란다면 8 번 비트를 설정할 것이다. 각 시스템 콜이 끝나서 제어권이 이를 부른 프로세스로 돌아가기 바로전에 `bh_active` 비트마스크를 검사하며, 만약 어떤 비트가 설정되어 있으면, 액티브로 표시된 하반부 핸들러 루틴들이 불린다. 비트 0 을 먼저 검사하고, 1 번을 다음에, 이런 식으로 31 번 비트까지 검사한다. 각 하반부 핸들러 루틴을 부르고 난 후에 `bh_active` 의 해당 비트는 0 으로 설정된다. `bh_active` 는 일시적인 것이다. 이는 단시 스케쥴러 호출 사이에만 의미가 있으며, 하반부 핸들러에서 더이상 할 일이 없을 때 이들을 부르지 않게 하는 방법이다.

역주 [108]) 아래에 나오는 하반부 핸들러는 각각 `TIMER_BH, CONSOLE_BH, TQUEUE_BH, NET_BH, IMMEDIATE_BH`로 정의되어 있다. 이들 외에도 다른 하반부 핸들러도 있으며, include/linux/interrupt.h에서 확인할 수 있다. (flyduck)

역주 [109]) 이는 원문의 내용이 틀린 것이라고 생각하지만, TQUEUE는 각 타이머 틱마다 활성화되는 하반부 핸들러로, `tq_timer` 작업큐를 처리하는 역할을 한다. 앞의 TIMER 하반부 핸들러 역시 각 타이머 틱마다 활성화되지만 11.3에 나오는 커널 타이머를 처리하는 역할을 한다. 하지만 둘 다 타이머 틱이 발생했을 때 활성화된다는 점은 동일하지만 맡은 역할은 다르다. (kernel/sched.c의 `do_timer()` 참조) (flyduck)

역주 [110]) 이는 `mark_bh()` 함수를 해당하는 하반부 핸들러 상수와 함께 부르면 된다. (flyduck)

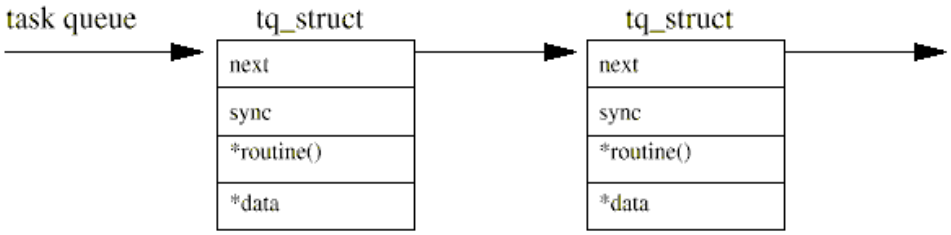


그림 11.2 : 작업 큐

11.2 작업큐(Task Queue)

작업큐는 커널이 작업을 나중으로 미루는데 사용하는 방법이다. 리눅스는 작업을 큐에 쌓아 두고 이를 나중에 처리할 수 있도록 하는 일반적인 메커니즘을 가지고 있다[111]. 작업큐는 종종 하반부 핸들러와 연결되어 쓰이기도 한다. 타이머 작업큐는 타이머 하반부 핸들러가 실행될 때 처리된다[112]. 작업큐는 그림 11.2 에서 보는 것과 같이, 함수의 주소와 다른 데이터를 가리키는 포인터를 가진 `tq_struct` 자료구조의 단일 연결 리스트로 이루어진 아주 간단한 자료구조이다. 작업큐에 있는 한 원소가 처리가 될 때 데이터 포인터와 함께 여기에 지정된 함수가 불린다.

커널에 있는 어떤 것이든 (예를 들어 디바이스 드라이버같은) 작업큐를 만들고 사용할 수 있지만, 실제로 커널이 만들고 관리하는 작업큐로는 다음 세가지가 있다[113].

타이머(timer) 이 큐는 다음 시스템 클럭 틱이 발생하였을 때 가능한 빨리 처리되어야 하는 일들을 큐에 넣기 위해서 사용된다. 각 클럭 틱이 발생할 때마다 여기에 무언가 있는지 검사하며, 이 큐에 무언가 있다면 타이머 큐 하반부 핸들러[114]가 액티브 상태로 바뀌게 된다. 타이머 큐 하반부 핸들러 역시 다른 하반부 핸들러와 마찬가지로 스케쥴러가 다음에 실행될 때 처리가 된다. 이 큐는 훨씬 복잡한 구조를 가지고 있는 시스템 타이머하고 혼동하지 말아야 한다[115].

즉시실행(immediate) 이 큐 역시 스케쥴러가 액티브 하반부 핸들러를 처리할 때 같이 처리된다. 즉시실행 하반부 핸들러는 타이머 큐 하반부 핸들러보다 우선순위가 낮으므로 이보다는 더 늦게 실행이 된다.

역주 [111]) 이런 용도로 앞에 하반부 핸들러를 설명했는데, 둘의 역할은 비슷하지만 메커니즘과 사용하는 경우는 서로 다르다. 하반부 핸들러는 한정된 자원인 반면에, 작업큐는 작업의 목록을 연결 리스트로 가지고 있으며, 별도의 작업큐를 정의하여 사용할 수 있기 때문에 확장이 가능하다. 작업큐는 타이머같이 타이머 인터럽트가 발생했을 때 처리될 작업 목록을 쌓아두기 위해서 사용되기도 하고, 디바이스 드라이버에서 작업을 미루기 위해서 사용한다. 모듈로 만들어진 디바이스 드라이버는 하반부 핸들러를 사용할 수 없으며, 작업큐 메커니즘을 사용해야 한다. (flyduck)

역주 [112]) 작업큐를 처리하는 함수는 `run_task_queue()`이며, kernel/sched.c에서 보면 `schedule()` 함수에서 `run_task_queue(&tq_scheduler)`를 부르며, `TQUEUE` 하반부 핸들러에서 `tq_timer`를, `IMMEDIATE` 하반부 핸들러에서 `tq_immediate`를 처리하는 것을 볼 수 있다. (flyduck)

역주 [113]) 아래 나오는 세가지 작업큐 외에 `tq_disk`가 있지만 이는 메모리 관리 서브시스템에서 내부적으로 사용하는 것이며, 다른 부분에서 사용할 수 없는 것이다. 이 세 작업큐는 각각 `tq_timer`, `tq_immediate`, `tq_schedule`로 정의되어 있다. (flyduck)

역주 [114]) 이 타이머 큐 하반부 핸들러는 앞에서 이야기한 바와 같이 `TQUEUE_BH`이다. (flyduck)

역주 [115]) 시스템 타이머는 `TIMER_BH`에서 처리하는 11.3장에서 설명할 타이머를 말한다. (flyduck)

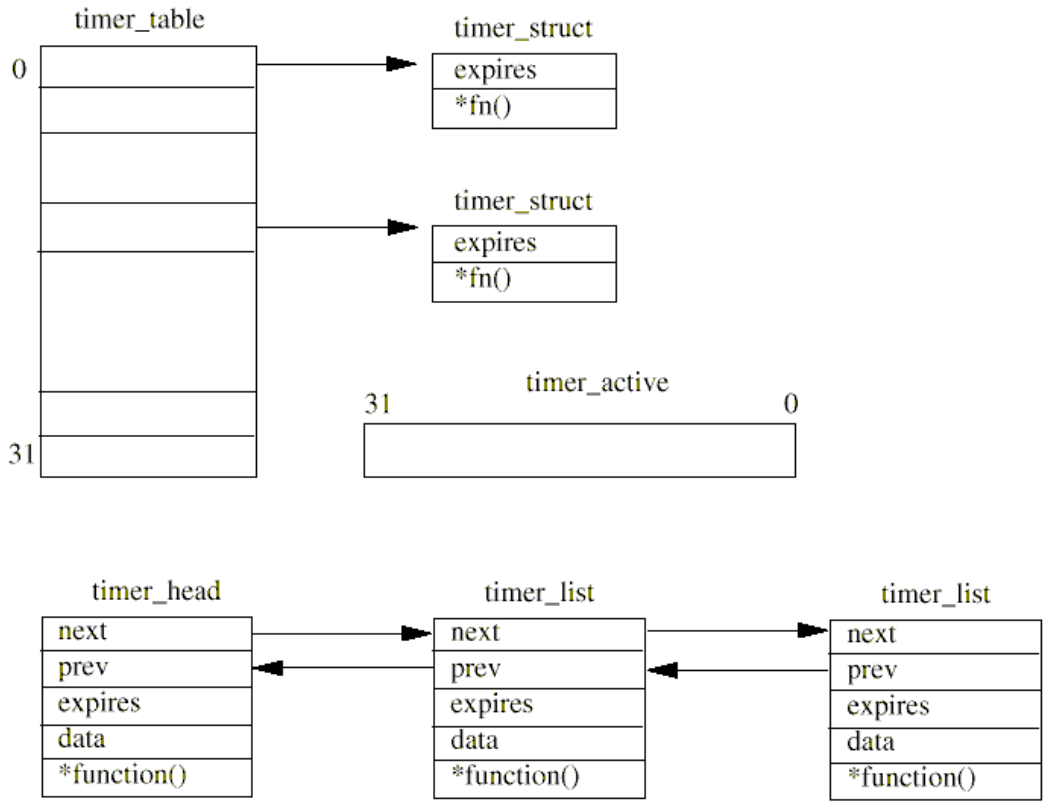


그림 11.3 : 시스템 타이머

스케쥴러(scheduler) 이 큐는 스케쥴러에 의해 직접 처리된다. 이는 시스템에 있는 다른 작업큐를 지원하기 위해서 사용되며, 이 경우 실행되는 작업은 디바이스 드라이버같은 것들을 위한 작업큐를 처리하는 루틴일 것이다.

작업큐가 처리되면 큐에 있는 첫번째 원소에 대한 포인터는 큐에서 제거되어 null 포인터로 바뀐다. 사실, 이 제거하는 과정은 하나의 쪼개질 수 없는 연산으로 처리되며, 중단될 수 없는 것이다. 이렇게 큐에 있는 각각의 원소들에 등록된 처리 루틴들이 차례로 호출이 된다. 큐에 있는 각 원소는 종종 정적으로 데이터를 할당받기도 한다. 그런데 여기에는 할당된 메모리를 알아서 해제하는 메커니즘이 본래 포함되어 있지 않다. 작업큐를 처리하는 루틴은 단지 리스트의 다음 원소로 이동할 뿐이다. 할당받은 커널 메모리를 제대로 해제하는 것은 큐에 있던 작업이 해야 할 일이다.

11.3 타이머(Timer)

운영체제는 미래의 어떤 시간에 해야할 행동들을 스케쥴할 수 있는 능력을 필요로 한다. 이들 행동들을 상대시간으로 정확하게 얼마간의 시간 후에 실행하도록 스케쥴하기 위한 메커니즘이 필요하다. 운영체제를 지원하기를 바라는 마이크로프로세서들은, 반드시 정기적으로 프로세서에게 인터럽트를 발생하는 프로그래밍 가능한 간격 타이머(interval timer)를 가지고 있어야 한다. 이렇게 정기적으로 발생하는 인터럽트를 시스템 클럭 틱(clock tick)이라고 하며, 이는 시스템의 행동들을 결집시키는 메트로놈과 비슷한 일을 하는 것이다. 리눅스는 현재 시간을 아주 단순하게 표현한다. 리눅스는 시간을 시스템이 부팅한 때부터 발생한 클럭 틱의 횟수 단위로 표현한다. 모든 시스템 시간은 이 단위로 되어 있으며, 이는 `jiffies` 라고 하며, 이와 똑같은 이름의 전역 변수가 존재한다[116].

역주 [116]) 이 `jiffies` 단위의 시간이 정확히 어느정도의 시간인지는 시스템마다 다르다.

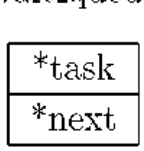


그림 11.4 : 대기큐

리눅스는 두가지 형태의 시스템 타이머를 가지고 있으며, 이 두 큐의 루틴들은 똑같은 시스템 타임에 호출되지만[117], 구현방식에 있어서 약간의 차이가 있다. 그림 11.3 은 이 두가지 메커니즘을 보여준다. 앞의 것은 예전의 타이머 방식으로서, 정적변수로 `timer_struct` 자료구조에 대한 포인터 32 개를 배열로 가지고 있으며, 액티브 타이머의 마스크인 `timer_active` 를 가지고 있다. 타이머가 타이머 테이블에 들어가는 것은 정적으로 정의된다 (하반부 핸들러 테이블인 `bh_base` 에 더 가깝다[118]). 각 항목들은 시스템 초기화 때 대부분 이 테이블에 추가된다. 두번째 방식은 더 새로운 것으로서 `timer_list` 자료구조를 만료시간의 올림순으로 가지고 있는 연결 리스트를 사용한다.

kernel/sched.c timer_bh() 참조

include/linux/timer.h 참조

kernel/sched.c run_old_timer() 참조

kernel/sched.c run_timer_list() 참조

이 두가지 방식 모두 만료시간을 `jiffies` 단위로 가지고 있는 시간을 이용하므로, 5 초 후에 실행되길 바라는 타이머라면, 5 초를 `jiffies` 단위로 변환한 후 현재 시스템 시간에 더하여 만료시간을 시스템 시간의 `jiffies` 로 나타내야 한다[119]. 모든 시스템 클럭 틱마다 타이머 하반부 핸들러는 액티브로 표시되고, 다음에 스케쥴러가 실행될 때 타이머 큐가 처리될 수 있도록 한다. 타이머 하반부 핸들러는 이 두가지 방식의 시스템 타이머를 모두 처리한다. 예전 방식의 시스템 타이머에 대해서는 `timer_active` 비트마스크를 검사하여 설정이 되어 있는 비트를 검사하게 된다. 만약 현재 액티브한 타이머의 만료시간이 지나면 (만료시간이 현재 시스템의 `jiffies` 보다 작으면), 타이머 루틴이 호출되고, 액티브 비트는 지워지게 된다. 새로운 방식의 타이머에서는, `timer_list` 자료구조의 연결 리스트에 있는 각 원소를 검사하여, 만료된 모든 타이머들은 리스트에서 제거되고, 등록된 함수가 호출된다. 새로운 타이머 방식은 타이머 루틴에 인자를 넘길 수 있다는 장점이 있다.

11.4 대기큐(Wait Queue)

프로세스가 시스템 자원을 기다려야 하는 경우가 많이 있다. 예를 들어, 프로세스가 파일 시스템에 있는 한 디렉토리를 나타내는 VFS inode 를 필요로 하는데 이 inode 가 버퍼 캐쉬에 있지 않은 경우, 프로세스는 파일 시스템을 가지고 있는 물리적인 장치에서 그 inode 를 가져오는 것을 기다려야 한다.

include/linux/wait.h 참조

리눅스 커널은 대기큐(그림 11.4 을 보라)라는, 프로세스의 `task_struct` 에 대한 포인터와 대기큐에 있는 다음 원소에 대한 포인터를 가지고 있는, 아주 단순한 자료구조를 사용한다.

---

arch/*/param.h에 `HZ`라는 상수가 정의되어 있는데, 클럭 틱은 초당 이 `HZ` 횟수만큼 발생하므로 1 `jiffie` = 1 / `HZ` 초라고 할 수 있다. 현재 커널에서 `HZ`는 알파 시스템에서는 1024로 다른 시스템에서는 100으로 정의되어 있다. 이 값을 바꾸어서 컴파일 할 수 있는데, 이 값이 커지면 시스템의 속도는 느려지겠지만 반응 속도는 더 빠를 것이며, 값이 작아지면 속도는 빨라지지만 반응 속도는 더 느려지게 된다. (flyduck)

역주 [117]) 예전의 타이머는 `run_old_timer()`에서, 새로운 타이머는 `run_timer_list()`에서 처리하며, 둘 다 `timer_bh()`에서 불린다. kernel/sched.c 참조 (flyduck)

역주 [118]) 구현방식으로 본다면 예전의 타이머는 정적으로 정의되고 부팅시에 핸들러가 등록되는 하반부 핸들러와, 새로운 타이머는 동적으로 사용하는 작업큐와 비슷하다고 할 수 있다. (flyduck)

역주 [119]) 즉 `jiffies`(현재 시간을 나타내는 전역변수) + 원하는 간격 * `HZ`로 계산한다. (flyduck)

프로세스가 대기큐의 끝에 추가가 되면, 이들은 인터럽트 가능(interruptible), 또는 인터럽트 불가능(uninterruptible) 상태가 된다. 인터럽트 가능한 프로세스는 대기큐에 있는 동안 발생하는 타이머 만료나 시그널같은 이벤트들에 의해서 인터럽트가 될 수 있다. 대기중인 프로세스의 상태는 이를 반영하여 `INTERRUPTIBLE` 또는 `UNINTERRUPTIBLE` 둘 중의 하나가 될 것이다. 이 프로세스는 지금 당장 계속 실행할 수 없기 때문에 스케쥴러가 실행되어, 새로 실행할 프로세스를 선택하게 되면 대기 프로세스는 중단이 된다[120].

대기큐가 처리가 될 때[121] 대기큐에 있는 모든 프로세스들의 상태는 `RUNNING`으로 바뀌게 된다. 만약 그 프로세스가 실행큐에서 제거된 것이었다면 다시 실행큐에 넣게 된다. 다음에 스케쥴러가 실행될 때 대기큐에 있던 프로세스들은, 더 이상 기다리고 있는 것이 아니기 때문에 실행될 수 있는 후보가 된다. 대기큐에 있는 프로세스가 스케쥴이 되면 제일 먼저 하는 일은 자신을 대기큐에서 제거하는 것이다. 대기큐는 시스템 자원에 대한 접근을 동기화하는데 사용할 수 있고, 리눅스가 세마포어를 구현하는데에도 사용한다. (아래를 보라)

11.5 버저락(Buzz Lock)

이것은 스핀락(spin lock)이라고 더 잘 알려져 있는데, 자료구조나 코드의 한 부분을 보호하는 가장 기본적인 방법이다. 이것은 코드의 임계지역 안에서 동시에 하나의 프로세스만 있도록 허용한다. 리눅스에서는 하나의 정수 항목을 락으로 사용하여 자료구조에 있는 항목에 대한 접근을 제한하는 목적으로 사용한다[122]. 임계지역으로 들어가고자 하는 각 프로세스들은 락의 초기값을 0 에서 1 로 바꾸려고 한다. 만약 현재 값이 1 이라면 프로세스는, 코드의 루프 안에서 계속 빙글빙글 돌면서 다시 시도하게 된다. 락을 가지고 있는 메모리 위치에 대한 접근은 반드시 한번에 이루어져야 한다(atmoic). 값을 읽고 그 값이 0 인지 확인하고, 0 이면 값을 1 로 바꾸는 것은 다른 어떤 프로세스에 의해서 중단되어선 안된다. 대부분의 CPU 구조들은 이를 특별한 명령어로 지원하지만, 캐시되지 않은 메인 메모리를 이용하여 버저락을 구현할 수도 있다.

락을 소유하고 있던 프로세스가 코드의 임계지역을 벗어날 때 버저락의 값을 감소시켜 0 이 되게 한다. 락을 검사하며 계속 돌고 있던 어떤 프로세스든지 이 값이 0 인 것을 알 수 있겠지만, 처음 읽은 프로세스가 이를 1 로 증가하고 임계지역으로 들어가게 될 것이다.

11.6 세마포어(Semaphore)[123]

---

[120]) REVIEW NOTE : 다음번에 스케쥴러가 실행될 때 INTERRUPTIBLE 상태에 있는 태스크가 실행되는 것을 막는 것은 무엇인가? 대기큐의 프로세스는 깨어날 때까지 절대로 실행되지 않는다.

역주 [121]) 대기큐가 처리가 될 때라는 것은, 기다리고 있던 자원을 사용할 수 있게 되어 이 자원을 기다리는 대기큐를 처리할 때라는 것이다. (flyduck)

역주 [122]) 다음 세마포어에서 이 스핀락을 `semaphore` 자료구조의 `waking` 항목에 대한 접근을 제어할 때 사용하는 것을 볼 수 있다. (flyduck)

역주 [123]) 이 세마포어는 IPC에서 나온 세마포어와 다르다. 이 세마포어는 SMP에서 한 프로세서만이 커널 모드로 들어갈 수 있도록 사용하는 것이다. 리눅스에서 SMP는 현재 효율적으로 만들어지지 않았다. 리눅스 커널은, 커널 모드에서 자신이 제어권을 놓지 않는 한 다른 프로세스에 의해 중단되지 않으며, 인터럽트 처리 루틴도 자신보다 높은 우선순위를 가진 인터럽트가 아닌 다른 프로세스에 의해 중단되지 않는다는 가정을 가지고 있다. 즉 커널모드에서 자료구조를 수정하는 것이 다른 것에 의해 중단되지 않는다는 가정을 가지고 있는 것이다. 이는 SMP에서 문제가 되는데, 왜냐하면 한 프로세서에서 커널 모드로 들어가 자료구조를 수정하고 있을 때, 다른 프로세서에서 커널모드로 들어가면 커널이 유지하는 자료구조를 동시에 여러 프로세서가 수정하게 되기 때문이다. 이의 가장 올바른 해결책은 당연히 자료구조를 수정하기 전에 임계지역을 표시하고 다른 프로세서가 접근하지 못하게 하는 것이지만, 이는 현재 구조상 너무 방대한 작업을 필요로 한다. 그

세마포어는 코드나 자료구조의 임계구역을 보호하는데 사용된다. 디렉토리를 나타내는 VFS inode 같은 임계 자료에 접근하는 것은, 프로세스의 다른 한 편에서 돌아가는 커널 코드에 의해서 이루어진다. 한 프로세스가 사용하고 있는 이런 중요한 자료구조를 다른 프로세스에서 고칠 수 있게 하는 것은 매우 위험하다. 이런 목적을 달성할 수 있는 한 방법은 임계자료에 접근하는 곳 주위에 버저락을 사용하는 것이지만, 이는 그다지 시스템 효율성이 좋지 않은 단순한 접근 방법이다. 대신 리눅스는 동시에 한 프로세스만이 코드나 데이터의 임계구역에 접근할 수 있도록 세마포어를 사용한다. 이 구역에 접근하려는 다른 모든 프로세스는 이 세마포어가 해제될 때까지 기다리게 될 것이다. 대기하게 되는 프로세스는 중단되지만, 시스템의 다른 프로세드들은 정상적으로 계속 동작할 수 있다.

리눅스 `semaphore` 자료구조는 다음과 같은 정보를 가지고 있다[124].

include/asm/semaphore.h 참조

카운트(count) 이 항목은 이 자원을 사용하려고 하는 프로세스들의 갯수를 관리한다. 양수는 이 자원이 사용가능하다는 것을 의미한다. 음수 또는 0 은 프로세스들이 그것이 해제되기를 기다리고 있다는 것을 의미한다. 초기값으로 1 을 주는 것은 단지 동시에 한 프로세스만이 이 자원을 사용할 수 있다는 것을 말한다. 프로세스가 자원을 얻고자 하면 카운트를 1 감소시키고, 자원의 사용이 끝나면 카운트를 1 증가시킨다.

깨울(waking) 이 자원을 기다리고 있는 프로세스의 수이며, 이 자원이 해제될 때 깨어나게 될 프로세스의 수이기도 하다.

대기큐(wait queue) 프로세스가 어떤 자원을 기다리면 그 자원의 대기큐에 들어간다.

락(lock) `waking` 항목을 접근할 때 사용하는 버저락이다.

세마포어의 초기 카운트가 1 이라고 할 때, 처음 사용하는 프로세스는 그 값이 양수라는 것을 알고, 1 을 감소시켜 0 으로 만든다. 이 프로세스는 이제 세마포어에 의해 보호되는, 코드나 자원의 임계부분을 "소유"하게 된다. 프로세스가 임계지역을 벗어나게 되면 세마포어의 카운트를 증가시킨다. 가장 최선인 경우는 임계지역을 소유하고자 하는 다른 프로세스가 없는 경우이다. 리눅스의 세마포어는 이 경우(가장 흔한 경우이기도 하다)에 대해 효율적으로 동작하도록 구현되었다[125].

만약 다른 프로세스가 소유하고 있는 임계지역에 한 프로세스가 들어가려고 할 때, 이 프로세스도 역시 카운트를 1 감소시킨다. 이번엔 카운트가 음수(-1)이므로 프로세스는 임계지역에 들어가지 못한다. 대신 영역을 소유하고 있는 프로세스가 영역을 빠져나갈 때까지 기다려야 한다. 리눅스에서는 기다리는 프로세스를 재우고, 임계지역을 소유하고 있는 프로세스가 임계지역을 빠져나갈 때 이를 깨우도록 한다. 기다리는 프로세스는 자신을 세마포어에 있는 대기큐에 추가하고, 루프를 돌면서 `waking` 항목의 값을 검사하고, `waking` 이 0 이 아닌 값이 될 때까지 스케쥴러를 호출하는 일을 반복한다[126].

래서 현재 SMP 구현은 하나의 세마포어를 사용하여 동시에 한 프로세서만이 커널모드에 있을 수 있게 하며, 이 장에서 설명하는 세마포어는 이런 용도를 위해 사용하는 것이다. 그래서 SMP에서도 커널 모드에서 동작하는 프로세스가 다른 프로세스에 의해 중단되지 않게하는 것이다. 이는 커널 모드에서 잡아먹는 CPU 시간이 전체 시스템 효율성의 병목으로 작동하게 되며, 커널 모드에 많이 진입하는 I/O 중심의 시스템에서는 더욱 병목현상이 더 심해지게 된다. 앞으로 효율적인 SMP 시스템을 구현하려면 필요한 경우에만 락을 걸 수 있도록 수정되어야 할 것이다. (flyduck)

역주 [124]) 이 책의 바탕인 2.0.33 소스에는 `lock` 항목이 있지만, 2.0.2x 버전이나 2.2.x 버전에서 `lock` 항목을 찾을 수 없다. (flyduck)

역주 [125]) 즉 한 프로세서만이 커널 모드에 진입할 때 가장 효율적으로 동작하도록 설계되었다는 뜻이다. (flyduck)

역주 [126]) 커널 모드에 진입하기 위하여 세마포어를 얻으려고 했는데 이를 얻을 수 없다면, 자신은 세마포어를 사용할 수 있게 될 때까지 스케쥴러를 호출하며(이것은 다른 프로세

임계지역의 소유자는 세마포어의 카운트를 증가시키는데, 그 값이 0 보다 작거나 같으면 잠들어서 이 자원을 기다리는 프로세스가 있다는 것이다. 가장 최선의 경우는 세마포어의 카운트가 다시 초기값인 1 이 되어서, 더이상 필요한 일이 없는 것이다. 소유하는 프로세스는 `waking` 카운터를 증가시키고, 세마포어의 대기큐에서 잠들어 있는 프로세스를 깨운다. 기다리는 프로세스가 깨어났을 때 `waking` 카운터는 이제 1 이 되어 있을 것이고, 이 프로세스는 이제 임계지역에 들어갈 수 있게 된다. 이 프로세스는 `waking` 카운터를 0 으로 감소시키고, 자신의 작업을 계속하게 된다. 세마포어의 `waking` 항목에 대한 접근은 세마포어의 락 항목을 이용한 버저락에 의해 보호된다.

번역 : 이호, 심마로
정리 : 이호

스가 자신 대신에 실행될 수 있게 만든다), 세마포어를 얻을 수 있을 때까지 기다린다는 것이다. (flyduck)

12 장

모듈 (Modules)

이 장에서는 리눅스 커널이 파일 시스템같은 함수들을 자신이 필요로 할 때 동적으로 로드하는 방법을 설명한다.

리눅스는 단일(monolithic) 커널이다. 즉 커널의 모든 기능적인 요소들이 자신의 내부 자료구조와 함수들에 모두 접근할 수 있는 하나의 거대한 프로그램이다. 운영체제 설계의 다른 방법으로는 커널의 각 기능적인 부분들이 별도의 단위로 쪼개지고, 그 사이에 엄격한 통신 매커니즘으로 연결되는 마이크로커널(micro-kernel) 구조가 있다. 이는 시간이 소모되는 프로세스[127]가 아닌 환경 설정 프로세스를 통하여 새로운 컴포넌트를 커널에 추가할 수 있게 한다. 가령 사용자가 NCR 810 SCSI 용 드라이버를 사용하려고 하는데 이것이 커널에 포함되어 있지 않다고 하자. 그러면 커널의 설정을 바꾸고 다시 컴파일해야 NCR 810 SCSI 를 사용할 수 있게 될 것이다. 그러나 여기에 다른 대안이 있다. 리눅스는 운영체제를 구성하는 컴포넌트들을 필요로 할 때 동적으로 로드 또는 언로드할 수 있게 한다. 리눅스 모듈은 시스템이 부팅된 후 언제라도 커널에 동적으로 링크될 수 있는 코드 덩어리이다. 또한 모듈이 더이상 필요하지 않을 때는 커널과의 연결을 해제하고 제거할 수 있다. 리눅스 커널의 상당수는 디바이스 드라이버와, 네트웍 드라이버나 파일시스템 같은 유사 디바이스 드라이버(pseudo device driver)이다.

사용자는 `insmod` 나 `rmmod` 같은 명령으로 리눅스 커널 모듈을 명확하게 로드 또는 언로드를 할 수 있으며, 또는 커널 자신이 자신이 필요로 할 때 커널 데몬(`kerneld`)에게 모듈을 로드/언로드 할 것을 요구할 수 있다. 필요로 할 때 코드를 동적으로 로드하는 것은 커널 크기를 최소화할 수 있고, 커널을 매우 유연하게 할 수 있어 매력적이다. 필자가 사용하는 인텔 커널은 모듈을 광범위하게 사용하여 크기가 겨우 406 Kbyte 밖에 되지 않는다. 나는 VFAT 파일 시스템을 가끔씩 사용할 뿐이므로, 내가 VFAT 파티션을 마운트 할 때만 리눅스 커널이 VFAT 파일 시스템 모듈을 자동으로 올리도록 했다. 그리고 그 VFAT 파티션의 마운트를 해제하면 시스템이 더이상 VFAT 파일 시스템 모듈이 필요하지 않다는 것을 알아차리고 시스템에서 제거하도록 했다. 모듈은 또한 새로운 커널 코드를 다시 컴파일하고 커널을 재부팅하지 않고 테스트를 해보고자 할 때 유용하다. 물론 아무런 댓가도 없는 것은 아니지만, 커널 모듈과 관련하여 성능과 메모리에서 약간의 손해가 있을 뿐이다. 이것은 로드할 수 있도록 모듈이 제공해야 하는 약간의 코드가 있고, 별도의 자료구조가 메모리를 조금 차지하기 때문이다. 또한 커널 자원에 접근할 때 한 단계를 거쳐야 하므로 모듈의 효율성이 아주 조금 떨어지게 된다.

로드된 리눅스 모듈은 다른 보통 커널 코드처럼 커널의 한 부분이 된다. 모듈은 커널 코드와 똑같은 권한과 책임을 진다. 다르게 말하면, 리눅스 커널 모듈은 모든 커널 코드나 디바이스 드라이버처럼 커널을 망가뜨릴 수도 있다는 것이다.

역주 [127]) 커널을 새로 컴파일하는 것을 가리킨다. (flyduck)

모듈이 자신이 필요로 할 때 커널의 자원을 사용할 수 있으려면, 그것이 어디 있는지 찾을 수 있어야 한다. 가령 모듈이 커널 메모리를 할당하는 함수인 kmalloc()을 호출해야 한다고 하자. 모듈을 컴파일할 때에는 메모리의 어느 위치에 kmalloc()이 있는지 모르므로, 모듈이 로드될 때 커널은 모듈이 제대로 동작할 수 있도록 kmalloc()에 대한 참조를 맞춰주어야 한다. 커널은 커널의 모든 자원의 목록을 커널의 심볼 테이블(symbol table)로 관리하며, 이를 이용해 모듈이 로드될 때 이들 자원에 대한 참조를 해결할 수 있다. 리눅스는 한 모듈이 다른 모듈의 서비스를 필요로 하는 경우, 모듈이 층층이 쌓아질 수 있도록 한다[128]. 예를 들어, VFAT 파일 시스템 모듈은 FAT 파일 시스템 모듈의 서비스를 필요로 한다. 이는 VFAT 파일 시스템이 FAT 파일 시스템을 다소 확장한 것이기 때문이다. 이렇게 한 모듈이 다른 모듈이 제공하는 서비스나 자원을 필요로 하는 것은, 모듈이 커널 자체의 서비스와 자원을 필요로 하는 경우와 매우 비슷하다. 단지 여기서 필요로 하는 서비스가 다른, 이전에 로드된 모듈에 있는 것일 뿐이다. 각 모듈이 로드될 때, 커널은 새로 로드되는 모듈에서 외부로 보여주는 자원과 심볼을 모두 커널 심볼 테이블에 추가한다. 이는 다음에 로드되는 모듈이 이미 로드된 모듈의 서비스를 이용할 수 있도록 하기 위한 것이다.

모듈을 언로드하려 할 때 커널은 모듈이 현재 사용되고 않고 있는지 알아야 하며, 모듈에게 자신이 언로드되려고 한다는 것을 알려줄 수 있어야 한다. 이렇게 해서 모듈은 커널에서 제거될 때, 자신이 할당받은 커널 메모리나 인터럽트 같은 시스템 자원을 해제할 수 있다. 모듈이 언로드될 때 커널은 모듈이 커널 심볼 테이블에 추가한 심볼들을 모두 제거한다.

로드된 모듈이 잘못 만들어진 것이어서 운영체제를 망가트릴 가능성과는 별도로, 다른 위험 가능성이 있다. 만약 지금 실행하고 있는 커널보다 이전 버전이나 이후 버전 용으로 컴파일된 모듈을 로드하려고 한다면 어떻게 될까? 모듈이 커널 루틴을 호출할 때 잘못된 인자를 넘겨준다면 문제가 생길 수 있을 것이다. 커널은 모듈을 로드할 때 엄격한 버전 검사를 하여 이런 문제를 선택적으로 막을 수 있다[129].

12.1 모듈을 로드하기

커널 모듈을 로드하는 방법은 두가지가 있다. 하나는 insmod 명령을 사용하여 수동으로 모듈을 커널에 추가하는 것이다. 두번째는 이보다 더 똑똑한 방법으로 모듈을 필요로 할 때 로드하는 것으로, 이를 요구시 로딩(demand loading)이라고 한다. 커널이 어떤 모듈을 필요로 한다는 것을 발견하면 (예를 들어 사용자가 커널에 없는 파일시스템을 마운트 한 경우), 커널은 커널 데몬(kerneld)에게 맞는 모듈을 로드하라고 요구한다.

kerneld 는 insmod, lsmod, rmmod 와 함께 모듈 패키지에

include/linux/kerneld.h 참조

커널 데몬은 비록 슈퍼유저 권한을 가지고 있기는 하지만 보통의 사용자 프로세스이다. 이 프로세스는 보통 시스템이 부팅할 때 시작하여, 커널과 프로세스간 통신(IPC) 채널을 하나 연다. 이 연결은 커널이 kerneld 에게 여러가지 작업을 요청하기 위해 메시지를 보내는데 사용한다. kerneld 의 주된 역할을 커널 모듈을 로드하고 언로드하는 것이지만, 필요할 때 직렬라인 상에 PPP 연결을 시작하거나, 필요하지 않을 때 이를 닫는 것 같은 다른 작업을 할 수 있는 능력도 있다. kerneld 는 직접 이런 일들을 하는 것이 아니라, 이런 일을 하기 위해 필요한 프로그램(insmod 같은 것)을 실행한다. kerneld 는 단지 커널의 대리인이며, 커널의 다른 한편에서 일을 스케쥴링한다.

역주 [128]) 이를 module stacking이라고 한다. (flyduck)

역주 [129]) 모듈을 컴파일할 때 커널의 버전 정보를 넣을 수 있다 이 경우 insmod가 모듈을 로드할 때 버전 검사를 하여, 버전이 맞지 않으면 모듈을 로드할 수 없다. insmod -f 옵션을 사용하면 버전이 맞지 않더라도 로드하게 할 수는 있지만 안전하진 않을 것이다. 좀 더 좋은 방법으로 모듈이 사용하는 커널 서비스에 넘겨주는 인자들이 달라진 경우에만 모듈을 로드할 수 없게 할 수 있다. 즉 컴파일 된 모듈이 사용하는 서비스가 현재 커널에서 제공하는 서비스와 달라진 것이 없다면 문제가 되지 않으며, 이를 위해선 커널과 모듈 양쪽에서 심볼이 인자정보를 체크섬으로 가지고 있도록 해야한다. (flyduck)

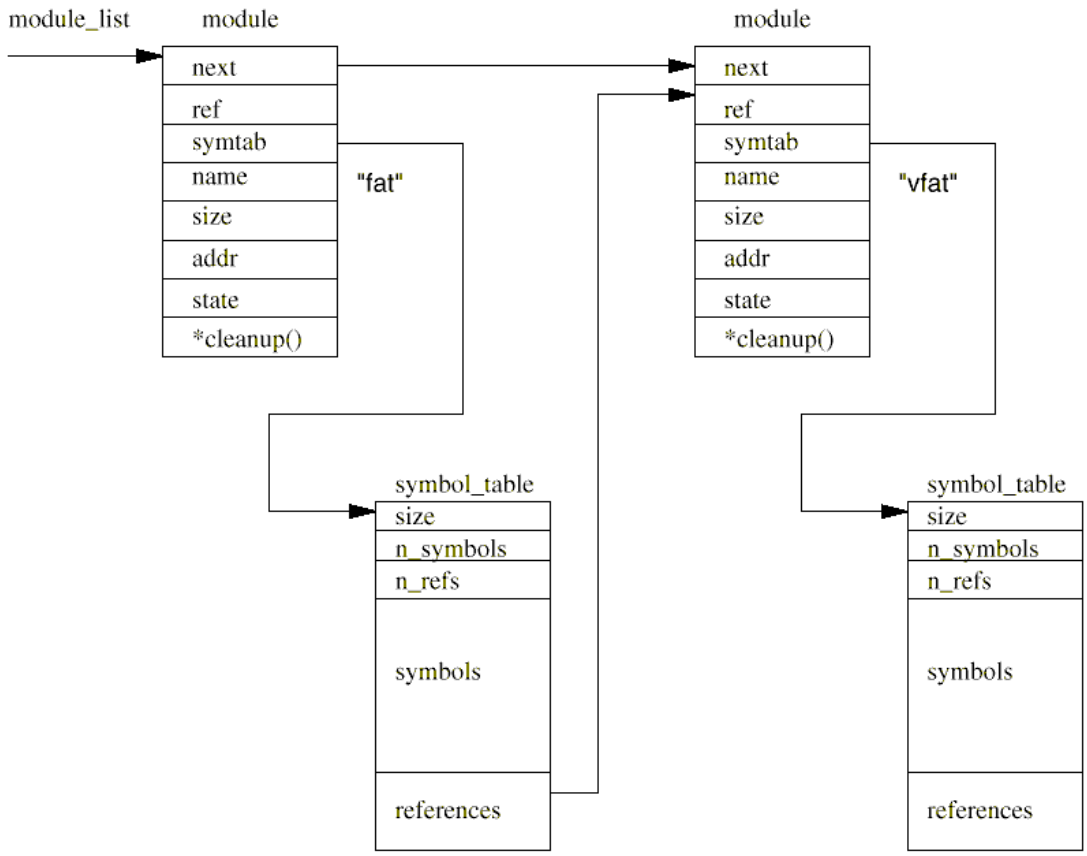


그림 12.1 : 커널 모듈의 리스트

insmod 프로그램은 자신이 로드해야 하는 요청한 커널 모듈을 찾을 수 있어야 한다. 요구시 로드하는 커널 모듈은 보통 /lib/modules/kernel-version 에 들어 있다. 커널 모듈은 시스템에 있는 다른 프로그램과 비교하면 링크된 오브젝트 파일이라는 점에 같지만, 재배치가능한 이미지로 링크되어 있다는 점이 다르다. 즉, 특정 주소에서 시작하도록 링크되어 있지 않다는 것이다. 이 이미지는 a.out 포맷이나 ELF 포맷의 오브젝트 파일일 수 있다. insmod 는 커널이 익스포트(export)하는 심볼을 찾기 위해 특권층의 시스템 콜을 사용한다. 커널은 익스포트 심볼을 심볼의 이름과 그것의 값(심볼의 주소같은)의 쌍으로 가지고 있다. 커널의 익스포트 심볼 테이블은, 커널이 관리하는 모듈의 목록인 module_list 포인터가 가리키고 있는, 첫번째 module 자료구조에 들어 있다. 커널에 있는 모든 심볼들이 모듈에게 익스포트 되는 것은 아니다. 단지 커널을 컴파일하고 링크할 때 특별히 지정한 심볼만이 이 테이블에 들어간다[130]. 드라이버가 시스템의 특정 인터럽트의 제어권을 갖고 싶을 때 호출해야 하는 커널루틴인 "request_irq" 심볼을 예로 들어보자. 필자가 갖고 있는 현재 커널에서 이것의 값은 *0x0010CD30* 이다. 커널의 익스포트 심볼과 값은 /proc/ksyms 를 살펴보거나 ksyms 프로그램을 사용하여 볼 수 있다. ksyms 프로그램을 이용하여 커널에 있는 모든 익스포트 심볼을 볼 수도 있고, 로드된 모듈이 익스포트하는 심볼들의 목록만 볼 수도 있다. insmod 는 모듈을 자신의 가상 메모리 공간으로 읽어들이고, 아직 해결되지 않은 커널 루틴과 자원에 대한 참조를 커널에 있는 익스포트 심볼을 통하여 맞추어준다. 이렇게 위치를 고정하는 것은 메모리상에 있는 모듈 이미지를 수정하는 형태로 이루어진다. insmod 는 모듈에 있는 해당하는 위치에 물리적으로 심볼의 주소를 써넣는다.

insmod 가 모듈의 익스포트된 커널 심볼에 대한 참조를 모두 해결하였다면, 특권 시스템 콜

역주 [130]) kernel/ksymc.c에 보면 커널이 익스포트할 심볼들의 목록이 들어 있다. 이와 마찬가지로 모듈을 만들 때 모듈에 있는 모든 심볼들을 익스포트하지 않고 필요한 것만 익스포트하도록 할 수 있다. 이는 너무 많은 심볼들이 심볼 테이블에 들어가 발생할 수 있는 문제를 미리 막기 위한 것이다. (flyduck)

kernel/module.c sys_create_module() 참조

kernel/module.c sys_get_kernel_sysm() 참조

include/linux/module.h 참조

을 이용하여 커널에게 새로운 커널을 포함할 수 있는 충분한 공간이 있는지 묻는다. 커널은 새 module 자료구조와, 새 모듈을 충분히 포함할 수 있는 크기의 커널 메모리를 할당하고, 이 구조체를 커널 모듈 리스트의 끝에 넣는다. 새 모듈은 초기화되지 않았다고 (UNINITIALIZED) 표시된다. 그림 12.1 은 FAT 와 VFAT 두 모듈이 커널에 로드된 후의 커널 모듈의 리스트를 보여준다. 이 그림에는 나타나지 않았지만, 리스트에 있는 첫번째 모듈은 유사 모듈(pseudo module)로서 단지 커널의 익스포트 심볼 테이블을 갖기 위해 존재한다. 로드된 커널의 목록과 그들의 상관관계를 보고 싶으면 lsmod 명령어를 쓰면 된다. lsmod 명령은 단지 커널 module 자료구조의 리스트로 부터 만들어지는 /proc/modules 의 포맷을 바꾸어서 보여주는 것 뿐이다. 커널이 모듈을 위해 할당한 메모리는 insmod 가 이에 접근할 수 있도록 insmod 프로세스의 주소공간에 매핑이 된다. insmod 는 모듈을 할당받은 공간으로 복사를 하고 이를 재배치하여, 할당받은 커널 공간에서 실행될 수 있도록 한다. 이는 모듈이 서로 다른 리눅스 시스템에서 똑같은 주소에 로드되거나 두번 모두 같은 주소에 로드된다는 보장이 없기 때문에 반드시 필요하다. 다시 한번, 이렇게 재배치하는 것은 모듈의 이미지를 올바른 주소로 수정하는 것을 포함한다.

kernel/module.c sys_init_module() 참조

새 모듈은 또한 커널에 심볼들을 익스포트하기 때문에, insmod 는 이렇게 익스포트된 이미지의 테이블을 만든다. 모든 커널 모듈은 모듈 초기화와 모듈 정리 루틴을 가지고 있어야 한다[131]. 이 두 심볼은 익스포트 되진 않지만, insmod 는 이 둘의 주소를 알아내어 커널에 넘겨야 한다. 모든 것이 잘 되었다면, insmod 는 이제 모듈을 초기화할 준비가 되어 있고, 특권 시스템 콜을 불러 커널에 모듈의 초기화 루틴과 정리 루틴의 주소를 넘긴다.

새 모듈이 커널에 추가되면, 커널의 심볼 목록을 갱신하고 새 모듈이 사용하는 모듈들을 수정해야 한다. 자신에 의존하는 다른 모듈을 가진 모듈은, 자신의 module 자료구조의 포인터가 가리키고 있는 자신의 심볼 테이블 끝에 참조되는 목록을 관리하여야 한다. 그림 12.1 은 VFAT 파일 시스템 모듈이 FAT 파일 시스템 모듈에 의존하고 있음을 보여준다. 따라서 FAT 모듈은 VFAT 모듈에 대한 참조를 포함하고 있다. 이 참조는 VFAT 모듈이 로드될 때 추가된 것이다. 커널은 모듈의 초기화 루틴을 부르고, 이것이 성공하면 모듈 설치를 계속 하게 된다. 모듈의 정리 루틴의 주소는 모듈의 module 자료구조에 저장되며, 모듈이 언로드될 때 커널에 의해 호출된다. 마지막으로 모듈의 상태는 실행중(RUNNING)으로 설정된다.

12.2 모듈을 언로드하기

모듈은 rmmod 명령을 사용하여 제거할 수 있지만, 요구시 로드된 모듈은 더이상 사용되지 않을 때 kerneld 에 의해 시스템에서 자동으로 제거된다. kerneld 의 타이머가 만료될 때마다, kerneld 는 사용되지 않는 요구시 로드된 모듈을 시스템에서 제거하는 시스템 콜을 부른다. 타이머의 값은 kerneld 를 시작할 때 설정되는데,. 필자의 시스템에서는 180 초마다 검사하도록 설정되어 있다. 그래서, 예를들어 ISO9660 파일시스템이 모듈로 되어 있는 곳에서 ISO9660 CDROM 을 마운트했다면, CDROM 을 언마운트한 후 조금 있으면 ISO9660 모듈이 커널에서 제거된다.

모듈은 커널의 다른 부분이 자신에 의존하고 있을 때에는 언로드될 수 없다. 예를 들어, 하나 이상의 VFAT 파일 시스템이 마운트되어 있는 동안에는 VFAT 모듈을 언로드할 수 없다. lsmod 의 출력을 눈여겨보면, 모듈에 숫자가 같이 붙어 나오는 것을 볼 수 있을 것이다. 예를 들어 :

```
Module:     #pages:     Used by
msdos          5              1
vfat           4              1 (autoclean)
fat            6        [vfat msdos]     2 (autoclean)
```

역주 [131]) 이들의 이름은 각각 init_module(), cleanup_module()로 정해져 있다. 이들은 심볼 테이블에 들어있지 않더라도 전역 함수로 되어 있다면 그 주소를 알아낼 수 있다. (flyduck)

카운트는 이 모듈에 의존하고 있는 커널 요소의 개수이다. 위의 예에서는, `vfat` 와 `msdos` 모듈이 `fat` 모듈에 의존하고 있으므로 카운트가 2 가 된다. `vfat` 와 `msdos` 모듈은 이 값으로 1 을 갖고 있는데 이것은 마운트된 파일시스템이다. 만약 다른 VFAT 파일 시스템을 읽어들이면, `vfat` 모듈의 카운트는 2 가 될 것이다. 모듈의 카운트는 그 이미지의 첫번째 longword 에 저장된다.

이 항목에는 또한 `AUTOCLEAN` 과 `VISITED` 플래그가 더 있다. 이 두 플래그는 요구시 로드된 모듈에서 사용된다. 이들 모듈은 자동으로 언도드 될 수 있다는 것을 시스템이 알 수 있도록 `AUTOCLEAN` 이라고 표시된다. `VISITED` 플래그는 모듈이 하나 이상의 다른 시스템 구성요소에 의해 사용되고 있음을 말한다. 이는 다른 구성요소가 그 모듈을 사용할 때마다 설정이 된다. `kerneld` 가 시스템에 사용되지 않고 있는 요구시 로드된 모듈을 제거하라고 요청할 때마다, 시스템은 자신에게 있는 모든 모듈을 뒤져서 그런 후보들을 골라낸다. 이는 단지 `AUTOCLEAN` 이라고 표시되어 있고 `RUNNING` 상태에 있는 모듈만을 찾는다. 만약 그 후보의 `VISITED` 플래그가 설정되어 있지 않다면 그 모듈을 제거하고, 그렇지 않다면 `VISITED` 플래그를 지우고 시스템의 다른 모듈을 계속 살펴본다.

한 모듈이 언로드 가능하다고 한다면, 그 모듈이 할당받은 커널의 자원을 해제할 수 있도록 모듈의 정리 루틴이 호출된다. 모듈의 자료구조는 `DELETED` 로 표시되고, 커널 모듈의 리스트와의 연결을 끊는다. 그 모듈이 의존하고 있는 다른 모듈은 더 이상 자신에 의존하지 않다는 것을 나타내도록 참조목록이 수정된다. 모듈이 필요로 했던 모든 커널 메모리는 해제된다.

kernel/module.c
sys_delete_module() 참조

번역 : 이호
정리 : 이호

13 장

프로세서 (Processors)

리눅스는 여러 프로세서에서 실행된다. 이 장은 이들을 간단히 설명한다.

13.1 X86

TBD

13.2 ARM

ARM 프로세서는 저전력 고성능의 32 비트 RISC 아키텍처이다. ARM 은 이동 전화, PDA(Personal Data Assistant)와 같은 임베디드(embedded) 장치에서 널리 사용되고 있다. ARM 은 31 개의 32 비트 레지스터를 가지고 있으며 각 모드에서 16 개를 사용할 수 있다. 명령어는 단순한 load 와 store 명령 (메모리에서 값을 가져오고, 계산하고, 결과를 메모리에 저장한다) 위주로 구성되어 있다. 한가지 재미있는 특징은 모든 명령이 조건부 명령이라는 것이다. 예를 들어 어떤 레지스터의 값을 테스트한 후, 다시 같은 조건을 테스트할 때까지, 테스트 결과에 따라 원하는 대로 조건부 명령을 실행할 수 있다. 또다른 재미있는 특징은 값을 메모리에서 로드하면서 산술/쉬프트 연산을 동시에 할 수 있다는 것이다. ARM 은 사용자 모드와, 여기서 SWI(소프트웨어 인터럽트)를 통해 들어갈 수 있는 시스템 모드를 포함하여 여러 모드에서 동작한다.

ARM 은 합성을 위한 핵심이며, ARM 사는 직접 프로세서를 생산하지는 않는다. 대신에 ARM 파트너(인텔이나 LSI 등)가 ARM 아키텍처를 실리콘으로 구현한다. 이 방식은 다른 프로세서가 보조프로세서 인터페이스를 통해 긴밀하게 결합될 수 있도록 하며, 여러 종류의 메모리 관리 유닛(memory management unit, MMU)의 변형을 갖고 있다. 이들은 단순한 메모리 보호 정책부터 복잡한 페이지 계층구조에까지 이른다.

13.3 알파 AXP 프로세서

알파 AXP 아키텍처는 64 비트 load/store RISC 아키텍처로서 속도를 염두에 두고 설계되었다[132]. 모든 레지스터는 64 비트로, 32 개의 정수 레지스터와 32 개의 실수 레지스터가 있다. 31 번 정수 레지스터와 31 번 실수 레지스터는 null 연산을 위해 사용된다. 이들 레지스터를 읽으면 0 이 돌아오고, 이들에 값을 쓰는 것은 아무런 효과도 없다. 모든 명령은 32 비트이며

역주 [132]) 높은 클럭을 제공할 수 있는 설계로 인해 21264이전에는 다른 RISC CPU에 비해 클럭당 성능 면에서는 좋지 못했다. (심마로)

메모리 연산은 읽기 아니면 쓰기이다. 이 아키텍처는 구현이 아키텍처를 따르는 한 여러가지 구현을 허용한다.

여기에는 메모리에 저장된 값을 직접 연산 대상으로 하는 명령은 없다. 모든 데이타 처리는 레지스터간에 이루어진다. 따라서 메모리의 카운터 값을 증가시키고 싶으면 먼저 레지스터에 읽어온 다음 값을 변경하고 메모리에 기록해야 한다. 명령들간의 상호작용은 한 명령이 값을 레지스터나 메모리에 쓰고, 다른 명령이 그 레지스터나 메모리에서 읽어오는 것을 통해 이루어진다. 알파 AXP 의 한가지 재미있는 특징은, 두 레지스터 값이 같은가를 테스트하는 것과 같이 플래그를 발생시키는 명령의 결과가, 프로세서 상태 레지스터에 저장되는 것이 아니라 제 3 의 레지스터에 저장할 수 있는 명령이 있다는 점이다. 처음 보기에는 이상하지만 상태 레지스터에 대한 의존을 제거함으로써 각 사이클에 여러 명령을 실행할 수 있는 프로세서를 만들기가 더욱 쉬워진다. 서로 관계없는 레지스터를 사용하는 명령은 하나의 상태 레지스터가 있을 때처럼 실행을 위해 서로를 기다릴 필요가 없다. 메모리에 대한 직접 연산이 없는 것과, 레지스터의 수가 많은 것도 여러 명령을 동시에 실행하는데 도움이 된다.

알파 AXP 아키텍처는 PALcode(특권 아키텍처 라이브러리 코드)라고 불리는 서브루틴들을 사용한다. PALcode 는 운영체제, 알파 AXP 아키텍처를 갖는 CPU 구현, 시스템 하드웨어에 따라 다르다. 이들 서브루틴은 컨텍스트 스위칭(context switching), 인터럽트, 예외(exception), 메모리 관리 등의 운영체제 프리미티브를 제공한다. 이들 서브루틴은 하드웨어나 CALL_PAL 명령에 의해 호출될 수 있다. PALcode 는 내부 프로세서 레지스터와 같은 저수준 하드웨어 기능에 대한 직접 접근을 제공하기 위해, 구현에 따른 약간의 확장을 포함한 표준 알파 AXP 어셈블러로 작성된다. PALcode 는 PALmode 에서 실행된다. 이 모드는 몇가지 시스템 이벤트의 발생을 중지시키고 PALcode 가 실제 시스템 하드웨어에 대한 제어를 완료하도록 하는 특권 모드이다.

번역 : 심마로
정리 : 이호

14 장

리눅스 커널 소스 (The Linux Kernel Sources)

이 장은 특정 커널 함수를 찾기 위해서 리눅스 커널 소스 어디서부터 시작해야 하는지 이야기한다.

이 책은 C 언어에 대한 지식을 요구하지는 않지만 리눅스 커널의 동작을 보다 잘 이해하려면 리눅스 커널의 소스를 가지고 있는 것이 좋다. 다시 말하면, 커널의 소스 프로그램은 리눅스 운영체제를 심도깊게 이해하는데 있어 효과적인 교재이다. 이 장은 커널 소스 전반에 대해 개괄한다. 즉 커널 소스가 어떻게 배열되어 있는지, 특정 코드를 찾으려면 어디서 시작해야 하는지 설명한다.

어디서 리눅스 커널 소스를 얻을 수 있는가

주요 리눅스 배포판들(Craftworks, Debian, Slackware, Red Hat 등)은 모두 리눅스 커널 소스를 포함하고 있다. 일반적으로 사용자의 리눅스 시스템에 설치된 리눅스 커널은 이 소스 코드를 컴파일하여 생성한 것이다. 리눅스의 성격상, 소스들이 계속 변경되므로 사용자의 시스템에 설치된 것은 조금 옛날 것이 되고 만다. 최신 버전의 소스 프로그램은 부록 B 에서 언급된 웹 싸이트에서 구할 수 있다. 이들은 `ftp://ftp.cs.helsinki.fi` 과 이를 그림자처럼 복사하는 다른 웹 싸이트에서 들어 있다. 헬싱키의 웹 싸이트가 가장 최신 버전의 소스를 가지고 있으며, MIT 나 Sunsite 와 같은 싸이트들로 비교적 최신 버전의 소스를 제공한다.

웹 싸이트에 접근할 수 없다고 하더라도, 많은 벤더들이 주요 웹 싸이트에 있는 내용들을 CD ROM 형태로 매우 저렴한 가격으로 제공하고 있으므로, 이를 이용하면 될 것이다. 1 년에 네번 혹은 매달 정기적으로 업그레이드판을 제공해주는 구독 서비스도 있다. 지역별 리눅스 유저 그룹도 소스를 구하는데 유용한 곳이다[133].

리눅스 커널의 버전 형태는 매우 단순하다. 짝수 버전 커널(예를 들자면 `2.0.30`)은 안정적이고 발표된 버전이고, 홀수 버전 커널(예를 들자면 `2.1.42`)은 모두 개발용 커널이다. 본책은 안정적인 2.0.30 소스 트리를 기반으로 하고 있다. 개발용 커널은 최신 기능들을 모두 포함하고 있으며 또한 최신 드라이버들도 모두 지원한다. 개발 커널은 불안정할 수도 있고, 이는 사용자가 바라지 않는 것이겠지만, 최신 커널을 사용해보는 것은 리눅스 공동체에 있어 중요한 일이다. 그래야 전체 공동체를 위해 테스트를 할 수 있다. 실제 제품으로 나온 커널이 아닌 것을 써보려고 할 때 시스템 전체를 백업해두는 것이 좋다는 것을 기억하기 바란다.

역주 [133]) 국내에서는 컴퓨터 잡지 부록의 형태도 큰 비중을 차지하고 있다.(심마로)

커널 소스에서 바뀐 것들은 패치(patch) 파일로 배포된다. patch 프로그램은 소스 파일들에 편집된 것들을 적용하는데 사용된다. 따라서, 예를 들어 2.0.29 커널 소스를 가지고 있고, 이를 2.0.30 소스로 바꾸고 싶다면, 2.0.30 패치 파일을 구해서 패치를 소스 트리에 적용하면 된다.

```
$ cd /usr/src/inux
$ patch -p1 < patch-2.0.30
```

이는 전체 소스 트리를 복사할 필요가 없어, 느린 직렬 연결을 통하는 경우 더욱 유용하다. 커널 패치를 구하기 좋은 곳은(공식적이던 비공식적이던) http://www.linuxhq.com 웹 사이트이다[134].

커널 소스는 어떻게 배열되어 있는가

소스 트리의 시작인 /usr/src/linux 에서 보면 여러개의 디렉토리가 있다.

arch arch 서브디렉토리는 모든 아키텍쳐에 종속적인 커널 코드를 포함하고 있다. 여기에는 서브디렉토리가 더 있는데, 각각 지원하는 아키텍쳐별로 있다. 예를 들어 i386, alpha 같은 이름의 서브디렉토리가 존재한다.

include include 서브디렉토리는 커널 코드를 빌드하는데 필요한 모든 인클루드(include) 파일들의 대부분을 가지고 있다. 여기에는 지원하는 아키텍쳐별로 하나씩 서브디렉토리가 있다. /include/asm 서브디렉토리는 현재 아키텍쳐에 필요한 실제 디렉토리로 (예를 들어, include/asm-i386) 소프트 링크되어 있다. 아키텍쳐를 다른 것으로 바꾸려면 커널 makefile 을 수정하고 리눅스 커널 환경설정 프로그램으로 돌아와야 한다.

init 이 디렉토리는 커널의 초기화 코드를 가지고 있으며, 커널이 어떻게 동작하는지 보기 시작하기에 좋은 곳이다.

mm 이 디렉토리는 모든 메모리 관리 코드를 가지고 있다. 아키텍쳐 종속적인 메모리 관리 코드는 arch/*/mm/ 아래에 있다. 예를 들어, arch/i386/mm/fault.c 같은 곳에 있다.

drivers 모든 시스템의 디바이스 드라이버는 이 디렉토리에 있다. 이들은 디바이스 드라이버의 유형별로 좀더 세분화 되면. 예를 들어 블럭 디바이스 드라이버는 block 에 있다.

ipc 이 디렉토리는 커널의 프로세스간 통신 코드를 가지고 있다.

modules 이는 단순히 빌드된 모듈을 저장하기 위한 디렉토리이다.

fs 모든 파일 시스템 코드를 가지고 있다. 파일 시스템별로 하나씩 디렉토리가 세분화된다. 예를 들어 vfat, ext2 같은 서브디렉토리가 있다.

kernel 메인 커널 코드가 들어 있다. 아키텍쳐 종속적인 커널 코드는 arch/*/kernel 에 있다.

net 커널의 네트워킹 코드가 들어 있다.

lib 이 디렉토리는 커널의 라이브러리 코드를 가지고 있다. 아키텍쳐 종속적인 라이브러리 코드는 arch/*/lib/에 있다.

scripts 이 디렉토리는 커널을 설정하는데 사용되는 스크립트(예를 들어 awk 나 tlk 스크립

역주 134) 인터넷 주소의 소유권 문제로 http://www.kernelnotes.com이 더 인정받고 있다. http://www.linuxhq.com은 갱신 빈도가 더 늦다.(심마로)

트)를 가지고 있다.

어디서부터 보기 시작할 것이가

리눅스 커널처럼 방대하고 복합적인 프로그램은 들여다보기에 위압적일 수 있다. 이는 실로 된 커다란 공처럼 끝이 보이지 않는 것이기도 하다. 커널의 한 부분을 보다 보면 관련된 다른 여러 파일들을 보게되고, 오래지 않아 무엇을 찾으려고 했는지 잊어버리게 된다. 다음 작은 장들은 어떤 주제를 보려 할때 소스 트리의 어디를 보는게 좋은지 힌트를 제공할 것이다.

시스템 시작과 초기화

인텔 기반 시스템에서, 커널은 loadlin.exe 나 LILO 가 리눅스 커널을 메모리로 읽어들인 후 커널에 제어권을 넘겨줌으로써 시작한다. 이 부분에 대해서는 `arch/i386/kernel/head.S` 를 보기 바란다. `head.S` 는 아키텍쳐 종속적인 셋업을 한 후 `init/main.c` 에 있는 `main()` 루틴으로 점프한다.

메모리 관리

이 코드는 대부분 `mm` 에 있지만, 아키텍쳐 종속적인 코드는 `arch/*/mm` 에 있다. 페이지 폴트 처리 코드는 `mm/memory.c` 에 있고, 메모리 매핑과 페이지 캐시 코드는 `mm/filemap.c` 에 있다. 버퍼 캐시는 `mm/buffer.c` 에, 스왑 캐시는 `mm/swap_state.c` 와 `mm/swapfile.c` 에 구현되어 있다.

커널

상대적으로 일반적인 코드는 `kernel` 에 있고, 아키텍쳐 종속적인 코드는 `arch/*/kernel` 에 있다. 스케쥴러는 `kernel/sched.c` 에 있고, *fork* 코드는 `kernel/fork.c` 에 있다. 하반부 핸들러 코드는 `include/linux/interrupt.h` 에 있다. `task_struct` 자료구조는 `include/linux/sched.h` 에서 찾을 수 있을 것이다.

PCI

PCI 유사 드라이버는 `drivers/pci/pci.c` 에 있고, 시스템 범위의 정의들은 `include/linux/pci.h` 에 되어 있다. 각 아키텍쳐들은 특정 PCI BIOS 코드를 가지고 있는데, 알파의 PCI BIOS 코드는 `arch/alpha/kernel/bios32.c` 에 있다.

프로세스간 통신

이것은 모두 `ipc` 에 들어 있다. 모든 시스템 V IPC 오브젝트들은 `ipc_perm` 자료구조에 들어 있고, `include/linux/ipc.h` 에서 찾을 수 있다. 시스템 V 메시지들은 `ipc/msg.c` 에, 공유 메모리는 `ipc/shm.c` 에, 세마포어는 `ipc/sem.c` 에 구현되어 있다. 파이프는 `ipc/pipe.c` 에 구현되어 있다.

인터럽트 처리

커널의 인터럽트 처리 코드는 대부분 모두 마이크로프로세서 (때때로 플랫폼) 종속적이다. 인텔의 인터럽트 처리 코드는 `arch/i386/kernel/irq.c` 에 있고, 정의는 `include/asm-i386/irq.h` 에 되어 있다.

디바이스 드라이버

리눅스 커널 소스 코드의 대부분은 디바이스 드라이버에 있다. 모든 리눅스 디바이스 드라이버 소스는 `drivers` 에 있지만, 이들은 장치 유형에 따라 세분화 된다.

/block 블럭 디바이스 드라이버. 예를 들어 IDE 디바이스 드라이버는 `ide.c` 에 있다. 모든 장치가 어떻게 파일 시스템을 가질 수 있으며, 어떻게 초기화되는지 보고 싶다면 `drivers/block/genhd.c` 에 있는 `device_setup()`을 보기 바란다. 이는 하드 디스크만 초기화하는 것이 아니라, 네트웍을 `nfs` 파일 시스템에 마운트하려고 한다면 네트웍도 초기화한다. 블럭 장치에는 IDE 와 SCSI 기반 장치가 포함된다.

/char `ttys`, 시리얼 포트나 마우스같은 문자 기반 장치들을 볼 수 있다.

/cdrom 리눅스의 모든 CDROM 코드가 들어 있다. 특별한 CDROM 장치(Soundblaster CDROM 같은)도 여기서 찾을 수 있다. IDE CDROM 드라이버는 `drivers/block` 에 있는 `ide-cd.c` 에 있고, SCSI CDROM 드라이버는 `drivers/scsi` 에 있는 `scsi.c` 에 있다는 점에 주의하기 바란다.

/pci 여기에는 PCI 유사 드라이버의 소스가 있다. PCI 서브시스템이 어떻게 매핑되고 초기화되는지 보기 좋은 곳이다. 알파 AXP PCI 확정 코드는 `arch/alpha/kernel/bios32.c` 에 있고, 이는 볼만한 가치가 있다.

/scsi 모든 SCSI 코드와 함께 리눅스가 지원하는 모든 SCSI 장치들의 드라이버가 있는 곳이다.

/net 네트웍 장치 디바이스 드라이버를 볼 수 있는 곳이다. DECChip 21040 PCI 이더넷 드라이버는 `tulip.c` 에 있다.

/sound 모든 사운드 카드 드라이버가 있는 곳이다.

파일 시스템

`EXT2` 파일 시스템 소스는 `fs/ext2/` 디렉토리에 있고 자료구조는 `include/linux/-ext2_fs.h`, `ext2_fs_i.h`, `ext2_fs_sb.h` 에 정의되어 있다. 가상 파일 시스템 자료구조는 `include/linux/fs.h` 에 정의되어 있고, 코드는 `fs/*`에 있다. 버퍼 캐시와 `update` 커널 데몬은 `fs/buffer.c` 에 구현되어 있다.

네트웍

네트워킹 코드는 `net` 에 있고, 인클루드(include) 파일들의 대부분은 `include/net` 에 있다. BSD 소켓 코드는 `net/socket.c` 에 있고, IP 버전 4 INET 소켓 코드는 `inet/ipv4/-af_inet.c` 에 있다. 일반적인 프로토콜 지원 코드는 (`sk_buff` 처리 루틴도 포함하여) `net/core/`에, TCP/IP 네트워킹 코드는 `net/ipv4/`에 있다. 네트워크 디바이스 드라이버는 `drivers/net` 에 있다.

모듈

커널 모듈 코드는 일부분은 커널에, 일부분은 `modules` 패키지에 있다. 커널 코드는 모두 `kernel/modules.c` 에 있고, 자료구조와 커널 데몬 `kerneld` 메시지는 `include/-linux/module.h` 와 `include/linux/kerneld.h` 에 있다. ELF 오브젝트 파일의 구조는 `include/linux/elf.h` 에서 볼 수 있다.

번역 : 이호, 이대현, 김진석, 심마로
정리 : 이호

용례 (Glossary)

인자 (Argument) 함수나 루틴에는 처리할 인자가 전달된다.

ARP Address Resolution Protocol, 주소 결정 프로토콜. IP 주소를 물리적 하드웨어 주소로 변환하는 데 사용한다.

아스키 (ASCII) American Standard Code for Information Interchange, 정보교환을 위한 미 표준 코드. 알파벳의 모든 글자는 8비트 코드로 표현한다. 아스키는 글자들을 저장하는 데 가장 많이 사용되는 코드이다.

비트 (Bit) 0이나 1(꺼진 상태와 켜진 상태)을 나타내는 한 비트의 데이터

하반부 핸들러 (Bottom Half Handler) 커널 내부에 있는 큐에 쌓인 작업을 수행하는 핸들러

바이트 (Byte) 8 비트의 데이터가 모여서 바이트를 이룬다.

C 고급 프로그래밍 언어의 일종. 리눅스 커널은 대부분 C로 되어 있다.

CISC Complex Instruction Set Computer, 복합 명령어 세트 컴퓨터. RISC의 반대 개념으로, 많은 수의 복합 어셈블리 명령어들을 지원하는 프로세서이다. X86 구조가 CISC 구조이다.

CPU Central Processing Unit, 중앙 처리 장치. 컴퓨터의 주처리부이다. 마이크로프로세서, 프로세서 참조

자료구조 (Data Structure) 여러 항목으로 이루어진, 메모리상에 있는 자료의 집합.

디바이스 드라이버 (Device Driver) 특정 장치를 제어하는 소프트웨어. 예를 들어, NCR 810 디바이스 드라이버는 NCR 810 SCSI 장치를 제어한다.

DMA Direct Memory Access, 직접 메모리 접근

ELF Executable and Linkable Format, 실행가능하고 링크할 수 있는 포맷. 이 오브젝트 파일 포맷은 유닉스 시스템 연구소(Unix System Laboritories)에서 개발되었으며, 이제는 리눅스에서 가장 일반적인 포맷으로 사용되고 있다.

EIDE Extended IDE, 확장 IDE

실행 이미지 (Executable Image) 기계어 명령과 데이터를 가지고 있는 구조화된 파일. 이 파일은 프로세스의 가상 메모리에 로드되어 실행할 수 있다. 프로그램 참조

함수 (Function) 어떤 동작을 수행하는 소프트웨어의 일부분. 예를 들어 두 값 중 큰 값을 돌려주는 함수 같은 것이다.

IDE Integrated Disk Electronics, 집적 디스크 전자장치

이미지 (Image) 실행 이미지를 보라.

IP Internet Protocol, 인터넷 프로토콜

IPC Interprocess Communication, 프로세스간 통신

인터페이스 (Interface) 루틴을 부르고 자료구조를 전달하는 표준 방식. 예를 들어 두 코드 계층 사이의 인터페이스는 특정 자료구조를 전달하고 돌려주는 루틴으로 표현할 수 있다. 리눅스의 VFS는 인터페이스의 좋은 예이다.

IRQ Interrupt Request Queue, 인터럽터 요구 큐

ISA Industry Standard Architecture, 산업 표준 구조. 이제는 좀 오래되긴 했지만, 플로피 디스크 드라이브같

은 시스템 구성요소를 위한 표준 데이터 버스 인터페이스이다.

커널 모듈 (Kernel Module) 파일 시스템이나 디바이스 드라이버같이 동적으로 로드할 수 있는 커널 함수

킬로바이트 (Kilobyte) Kbyte라고도 쓰며 1000 바이트를 뜻한다.

메가바이트 (Megabyte) Mbyte라고도 쓰며 1000000 바이트를 뜻한다.

마이크로프로세서(Microprocessor) 고밀도 직접된 CPU. 대부분의 요즘 CPU들은 마이크로프로세서이다.

모듈 (Module) 어셈블리어 명령어나 C같은 고급언어의 형태로 CPU 명령어들을 내장한 파일.

오브젝트 파일 (Object File) 실행 이미지로 만들어지기 전에, 다른 오브젝트 파일이나 라이브러리와 링크되지 않은, 기계어 코드와 데이터를 가지고 있는 파일.

페이지 (Page) 실제 메모리는 동일한 크기의 페이지들로 분할된다.

포인터 (Pointer) 메모리상의 다른 위치의 주소를 가지고 있는 메모리상의 한 위치 (변수)

프로세스 (Process) 프로그램을 실행할 수 있는 한 개체이다. 프로세스는 실행중인 프로그램으로 생각할 수 있다.

프로세서 (Processor) CPU와 같으며 마이크로프로세서를 줄인 말이다.

PCI Peripheral Component Interconnect, 주변장치 상호연결. 컴퓨터 시스템의 주변장치들을 연결하는 법을 설명한 표준.

주변장치 (Peripheral) 시스템에 있는 CPU를 위해 일하는 지능형 프로세서. IDE 컨트롤러 칩같은 것이 한 예이다.

프로그램 (Program) "hello world"를 출력하는 것처럼, 어떤 작업을 수행하는 통합된 CPU 명령어들의 집합. 실행 이미지 참고.

프로토콜 (Protocol) 프로토콜은 상호 협동하는 두 개의 프로세스나, 네트웍 계층 사이에 응용프로그램 데이터를 전송하는데 사용하는 네트웍용 언어이다.

레지스터 (Register) 정보나 명령어를 저장하는 데 사용되는, 칩 내의 한 위치.

레지스터 파일 (Register File) 프로세서 내의 레지스터 집합.

RISC Reduced Instruction Set Computer, 축소 명령어 세트 컴퓨터. CISC의 반대 개념으로, 어셈블리 명령어의 갯수가 적고, 각각의 명령어는 간단한 연산만 하는 프로세서이다. ARM과 알파 프로세서는 둘다 RISC 구조로 되어 있다.

루틴 (Routine) 함수하고 비슷하지만, 엄밀하게 말하면 루틴은 결과값을 돌려주지 않는다.

SCSI Small Computer Systems Interface, 소형 컴퓨터 시스템 인터페이스

쉘 (Shell) 운영체제와 사용자 사이에서 인터페이스 역할을 해주는 프로그램. 명령쉘(command shell)이라고도 한다. 리눅스에서 가장 보편적으로 사용하는 것은 bash 쉘이다.

SMP Symmetrical Multiprocessing, 대칭형 멀티프로세싱. 둘 이상의 프로세서가 일을 공정하게 나누는 시스템.

소켓 (Socket) 네트웍 연결의 한쪽 끝을 나타낸다. 리눅스는 BSD 소켓 인터페이스를 지원한다.

소프트웨어 (Software) CPU 명령어(어셈블러나 C같은 고급언어 모두)와 데이터. 대개 프로그램과 같은 의미를 갖는다.

시스템 V(System V) 1983년에 나온 유닉스의 변종. 무엇보다도 System V IPC 메커니즘을 포함하고 있다.

TCP Transmission Control Protocol, 전송 제어 프로토콜.

작업큐 (Task Queue) 리눅스 커널 내부에서 일을 연기하는데 사용하는 메커니즘.

UDP User Datagram Protocol, 사용자 데이터그램 프로토콜

가상 메모리 (Virtual Memory) 시스템에 있는 물리적인 메모리의 크기를 실제보다 크게 보이게 해주는 하드웨어적, 소프트웨어적 메커니즘.