

**Hanbit eBook**

**Realtime 37**

# 오픈 소스

Vol I

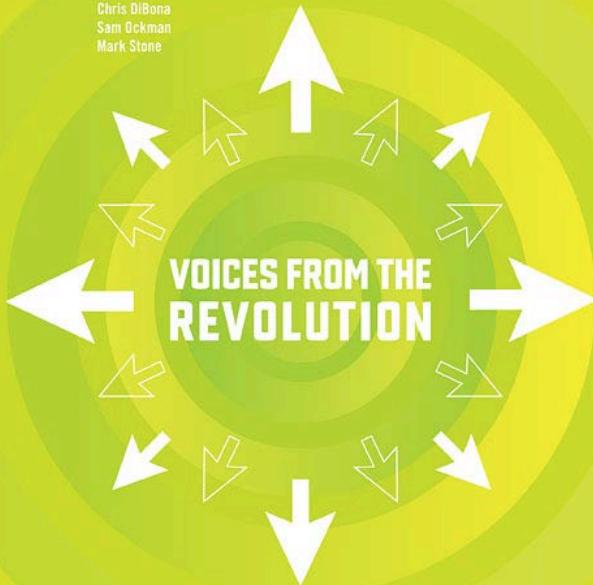
## 오픈 소스 혁명의 목소리

에릭 레이먼드, 리처드 스톤먼, 리누스 토르발스, 팀 오라일리 외 지음

송창훈, 이기동, 이만용, 최준호 옮김

# open sources

Chris DiBona  
Sam Ockman  
Mark Stone



이 도서는 O'REILLY의  
Open Sources의  
번역서입니다.

오픈 소스 혁명의 목소리

**오픈 소스**

**Vol. I**

## 오픈 소스 혁명의 목소리 **오픈 소스 Vol. I**

---

**초판발행** 2013년 7월 31일

**지은이** 에릭 레이먼드 외 17인 / **옮긴이** 송창훈, 이기동, 이만용, 최준호 / **펴낸이** 김태현

**펴낸곳** 한빛미디어(주) / **주소** 서울시 마포구 양화로 7길 83 한빛미디어(주) IT출판부

**전화** 02-325-5544 / **팩스** 02-336-7124

**등록** 1999년 6월 24일 제10-1779호

**ISBN** 978-89-6848-644-9 15000

**책임편집** 배용석 / **기획** 한빛미디어 스마트미디어팀 / **편집** 이종민, 정지연

**디자인** 표지 여동일, 내지 스튜디오 [밈]

**마케팅** 박상용

이 책에 대한 의견이나 오탈자 및 잘못된 내용에 대한 수정 정보는 **한빛미디어(주)**의 홈페이지나 아래 이메일로 알려주십시오.

**한빛미디어 홈페이지** [www.hanb.co.kr](http://www.hanb.co.kr) / **이메일** [ask@hanb.co.kr](mailto:ask@hanb.co.kr)

---

Published by HANBIT Media, Inc. Printed in Korea

Copyright © 2013 HANBIT Media, Inc.

Authorized translation of the English edition, © 1999 O'Reilly & Associates, Inc. This translation is published and sold by permission of O'Reilly & Associates, Inc., the owner of all rights to publish and sell the same.

이 책의 한국어판 저작권은 오라일리 사와의 독점 계약에 의해 **한빛미디어(주)**에 있습니다.

저작권법에 의해 보호를 받는 저작물이므로 무단 복제 및 무단 전재를 금합니다.

---

지금 하지 않으면 할 수 없는 일이 있습니다.

책으로 펴내고 싶은 아이디어나 원고를 메일([ebookwriter@hanb.co.kr](mailto:ebookwriter@hanb.co.kr))로 보내주세요.

한빛미디어(주)는 여러분의 소중한 경험과 지식을 기다리고 있습니다.

Individual contributors:

"A Brief History of Hackerdom" and "The Revenge of the Hackers" are Copyright © 1998 Eric S. Raymond. "The Open Source Definition" is Copyright © 1998 Bruce Perens. These essays are free; you can redistribute them and/or modify them under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the license, or (at your option) any later version. These essays are distributed in the hope that they will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

See Appendix B for a copy of the GNU General Public License, or write to the Free Software Foundation, Inc., 59 Temple Place – SUite 330, Boston, MA 02111–1307, USA.

Eric Raymond  
esr@thyrsus.com  
6 KarenDrive  
Malvern, PA 19355

Bruce Perens  
bruce@pixar.com  
c/o Pixar Animation Studios  
1001 West Cutting #200  
Richmond, CA 94804

"The GNU Operation System and the Free Software Movement" is Copyright ©1998 Richard M. Stallman. Verbatim copying and duplication is permitted in any medium provided this notice is preserved.

"Giving It Away" is Copyright ©1999 O'Reilly & Associates. All rights reserved to O'Reilly & Associates and Red Hat Software.

All other materials is Copyright ©1999 O'Reilly & Associates. All rights reserved.

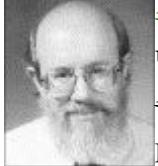
## 저자 소개



브라이언 벨렌도프 Brian Behlendorf는 보통 사람들이 생각하는 해커가 아니다. 그는 아파치 그룹의 부창립자이자 핵심 멤버이기 때문이다. 아파치는 공개적으로 접근 가능한 인터넷의 웹 서버 중 53%를 차지하는 오픈 소스 웹 서버다. 이 공개 프로그램은 마이크로소프트, 넷스케이프, 그리고 나머지 모든 벤더를 합한 것보다도 큰 시장을 점유한다. 브라이언은 4년간 아파치에 대한 작업을 해왔으며, 아파치 팀의 다른 멤버와 함께 프로젝트의 성장을 이끄는 데 도움을 주었다. 처음에는 단순히 흥미로운 실험으로 시작했던 것이 이제는 정교하게 만들어진 완전한 웹 서버가 되었다.

음악을 좋아하는 사람은 많겠지만 레이브 음악을 구성하거나 파티의 DJ를 하는 사람은 그가 유일한 것 같다. 그의 웹사이트인 <http://www.hypereal.org>는 홀륭한 음악과 레이브, 클럽의 리소스 사이트다. 그는 책 읽기를 좋아하며, 최근에는 컴퓨터 분야 외의 서적을 읽고 있다. 카프라Capra의 『물리학의 도 Tao of Physics』와 촘스키 Chomsky의 『비밀, 거짓말, 그리고 민주주의 Secrets, Lies and Democracy』는 특히 그가 즐겨 읽는 책이다.

1998년 후반, IBM은 하이엔드 AS/400라인에서 아파치를 지원한다고 발표했으며, 이는 아파치 프로젝트의 분수령을 이루는 사건이 되었다. 브라이언은 IBM의 움직임에 대해 “이것이 비즈니스가 될 수 있다고 생각한 사람이 나뿐만이 아니었다는 사실이 기쁘다. 즐겁게 일할 수 있을 뿐만 아니라 비즈니스 모델도 된다. 사람들은 오픈 소스가 실제로 컴퓨터를 위해 건강하고 이득이 되는 더 나은 길이 될 것이라는 사실을 알게 될 것이다.”라고 말했다.



스콧 브래드너 Scott Bradner는 아르파넷 ARPANET 초기 시절부터 하버드 대학에서 데이터 네트워크의 설계 및 운영에 관여해 왔다. 하버드 고속 데이터 네트워크 HSDN, 룽우드 의료단지 네트워크 LMANet, NEARNET 설계에 참여했으며, LMANet과 NEARNET, CoREN 기술위원회 초대 의장을 역임했다.

IETF 전송 분야 공동책임자와 IESG 운영위원을 맡고 있으며, ISOC 임원으로 선출돼 표준화 부문 부의장으로 활동하고 있다. 또한 IETF 차세대 IP 분야 운영위원으로 참여하고 있으며, 애디슨-웨슬리가 출판한 [『IPng: 차세대 인터넷 프로토콜』](#)을 공동 집필했다.

하버드 대학 수석 부총장실 Office of the Provost 선임 기술자문위원으로 학내 데이터 네트워크와 신기술에 대한 조언과 자문을 제공하며, 하버드 네트워크 장비 실험 연구소를 관리하고 있다. 많은 기술 콘퍼런스와 Interop 강사로, 「네트워크 월드」의 주간 칼럼니스트로 활약하고 있으며 네트워크 분야에 대한 독립적인 자문을 제공하기도 한다.



짐 하멀리 Jim Hamerly는 넷스케이프 커뮤니케이션사의 고객 제품부 부사장이다. 1997년 6월, 넷스케이프사는 짐이 창립자이며 사장이자 CEO였던 디지털 스타일사 Digital Style를 인수했다. 디지털 스타일사를 세우기 전에는 페이지스 소프트웨어사 Pages Software, Inc.의 엔지니어링 부사장이었으며, 여기서 그는 데스크톱 출판 도구인 Pages와 최초의 WYSIWYG 웹 저작 도구인 WebPages의 개발을 관리했다.

짐은 15년간 제록스Xerox의 여러 R&D와 제품 개발 활동을 했으며, 가장 최근에는 제록스사의 소프트웨어 부서인 XSoft의 차석 엔지니어로 4가지의 소프트웨어 제품 라인을 맡았었다. MIT, U.C. 버클리, 카네기 멜론 대학에서 각각 전자공학과 전산학 학사, 석사, 박사 학위를 취득했다.



**커크 맥퀴식**Kirk McKusick은 유닉스와 BSD 관련 주제에 대한 서적, 기사, 상담 자료 등을 집필하고 이에 대해 강의를 하고 있다. 캘리포니아 버클리 대학교에 있는 동안, 그는 4.2BSD 파일 시스템을 구현하였고, 4.3BSD와 4.4BSD의 개발과 배포를 조망하는 버클리 CSRG의 전산과학 연구원이었다. 그의 관심 분야는 가상 메모리 시스템과 파일 시스템이다. 그는 이 두 영역이 경계 없이 통합될 날을 고대하고 있다. 코넬 대학에서 전자공학 학사 학위를 취득했으며, 버클리 대학원에서 전산과학·경영학 석사학위, 전산 과학 박사 학위를 취득했다. 그는 Usenix 협회의 전 의장이었으며 ACM과 IEEE의 회원이기도 하다.

여가 시간에 수영과 스쿠버ダイ빙을 즐기며, 취미로 포도주 수집을 하고 있다. 그는 19년 이상 에릭 올만Eric Allman과 가족처럼 살고 있는데, 그와 공동으로 소유하고 있는 집 지하에 있는 특별히 건조된 와인 저장소에(<http://www.mckusick.com/~mckusick/index.html>에서 볼 수 있다) 수집된 와인을 보관하고 있다.



**팀 오라일리**Tim O'Reilly는 오라일리 미디어의 설립자이자 최고 경영자이며, 펄, 리눅스, 아파치와 같이 인터넷의 기반이 되는 오픈 소스 기술에 관계된 책을 출판하고 있다. 그는 ‘오픈 소스 정상 회의Open Source Summit’를 열어 주요 오픈 소스 공동체의 지도자들을 처음으로

한자리에 모았으며 저술과 강연, 회의를 통해서 오픈 소스 운동을 적극적으로 지원하고 있다. 또한 그는 Internet Society의 이사이기도 하다.



톰 페퀸 Tom Paquin은 처음에 병렬 프로세서와 관련된 프로젝트 작업을 위해 IBM 연구소에 입사했지만, 나중에 당시 새로운 PC를 위한 비트맵 그래픽 가속기(AMD 29116 기반)로 일을 마치게 되었다. MIT 와 브라운 대학에서 X6와 X9를 다루다 카네기 멜론 대학에서 최초의 상업적 X11 버전을 만들기 위한 작업을 도왔다.

그는 1989년 5월에 실리콘 그래픽스사 SGI에 입사했고, 여기서 GL과 X를 통합하는 작업을 했다. 1994년 4월에는 넷스케이프사의 짐 클락 Jim Clark와 마크 앤드레슨 Marc Andreessen과 합류했다. 그는 최고 엔지니어링 책임자로, 모질라의 1.0과 2.0 을 배포하는 동안 팀을 이끌었다. 현재 넷스케이프사 직원으로, Mozilla.org에서 관리자이자 문제 중재자, 신비한 정치적 지도자로 일하고 있다.



브루스 페렌스 Bruce Perens는 오랫동안 리눅스와 오픈 소스 소프트웨어를 지지해 온 인물이다. 그는 1997년까지 완전한 오픈 소스 소프트웨어 기반의 리눅스 배포판을 만들기 위해 여러 자원 봉사자들이 노력하는 프로젝트인 데비안 프로젝트의 리더였다. 그는 데비안 프로젝트 일을 하는 동안, 데비안 소셜 컨트렉트 Debian Social Contract를 다듬는 데 도움을 주었다. 데비안 소셜 컨트렉트는 데비안 배포판에 포함될 수 있는 자유 라이선스를 따르는 소프트웨어의 조건을 명시한 문서로 OSD Open Source Definition의 전신 前身이다. 데비안 리더로서의 활동을 그만둔 후, 그는 대중들의 이익을 위한 소프트웨어를 이끌고 만들고, 에릭 레이먼드와 함께 OSI Open Source Initiative를 만들어 오픈 소스 전파

를 위해 노력하고 있다. 그는 오픈 소스 소프트웨어 전파를 위한 활동을 하면서 틈틈이 Pixa Animation Studio에서 일한다.



에릭 스티븐 레이먼드 Eric Steven Raymond는 1970년 후반 아르파넷 시절부터 인터넷과 해커 문화에 매료되어 호기심을 갖고 관찰하고 참여해 온 해커다. 그는 컴퓨터에 매혹되기 전에 수학과 철학을 공부했고, (두 장의 앨범에서 플루트를 연주한) 음악가로도 어느 정도 성공을 거두었다. 그의 여러 오픈 소스 프로젝트는 주요 리눅스 배포판에 모두 포함되어 있는데, 가장 널리 알려진 것은 fetchmail이지만 GNU 이맥스와 ncurses에도 크게 공헌했다. 아무리 잘해도 감사받기는 힘든 일 중 하나인 termcap의 관리자다. 그는 태권도 검은띠 유단자며 사격으로 기분을 전환하기도 한다.

좋아하는 총은 클래식 1911년형 45구경 권총이다. 그는 『새로운 해커 사전』을 만들고 편집했으며 『GNU 이맥스 배우기』의 공동 저자이기도 하다. 1997년에 웹에 올린 「성당과 시장」이라는 제목의 글은 넷스케이프가 자사의 브라우저 소스 코드를 공개하도록 한 중요한 촉매제로 여겨진다. 그 시절부터 에릭은 오픈 소스 소프트웨어 물결을 솜씨 좋게 서평해 왔다. 최근 그는 리눅스와 오픈 소스 소프트웨어에 대해 마이크로소프트가 인식하고 있는 위협을 담은 일련의 내부 문서를 폭로했다. (최초 발견일 10월 31일이 할러윈데이여서 그렇게 이름 붙여진) 「핼러윈 문서」는 우스개 이야기 소재일 뿐 아니라 거대 소프트웨어 복합 기업이 오픈 소스 현상에 대해 보여준 최초의 확인된 반응이다.



이 책의 저자들은 모두 나름대로 리처드 매슈 스탈먼 Richard Matthew Stallman, RMS에게 빚을 지고 있다고 할 수 있다. 그는 15년 전에 자유 소프트웨어의 개발을 보호하고 진흥하기 위해 GNU 프로젝트를 시작했다. GNU 프로젝트가 표방했던 목표는 완전한 운영체제 전체와 GPL과 같은 이용허락으로 관리될 수 있는 완전한 도구들의 집합을 만들어 누구든지 무상으로 소프트웨어를 사용할 수 있도록 하려는 것이었다.

이맥스 편집기를 개발한 공로로 1991년 ACM이 주는 그레이스 호퍼상을 받았고, 1990년에는 맥아더 재단이 창의적인 개인에게 충분한 연구기회를 보장하기 위해 기금을 제공하는 맥아더상을 수상했다. 1996년에는 스웨덴 왕립 기술원에서 명예 박사 학위를 받았고 1998년에는 전자개척재단의 선구자상을 리누스 토르발스와 공동으로 수상했다. 현재 그는 그가 만든 소프트웨어보다 자유 소프트웨어 전도사로 더욱 널리 알려져 있다.

오픈 소스 운동에 헌신하는 다른 사람과 마찬가지로 RMS도 그가 속해 있는 공동체 안에서 논쟁을 불러일으키곤 한다. ‘오픈 소스 소프트웨어’라는 용어에 대해서, 그는 이 단어가 자유 소프트웨어가 가진 많은 측면 중에서 특히 자유와 관련된 부분을 없애기 위해 고안된 것이라고 주장한다. 이러한 유형의 주장은 그를 극단주의 자라고 부르는 사람이 생겨나게 하는 이유 중의 하나다. 그러나 RMS는 자신의 철학적 입장을 **이맥스 교의 성자 이그누시우스** St. IGNUcius의 모습으로 냉철히 지켜나가고 있다.

많은 사람은 “만약 스톤먼이 없었더라면 그를 만들어내야 했을 것이다”라고 말한다. 이런 사실은 리처드 스톤먼이 오늘날까지도 대중화하고 또한 전파하고 있는 자유 소프트웨어 운동이 없었다면 오픈 소스 운동이 결코 있을 수 없었으리라는 사실에 대한 진심 어린 감사의 표현일 것이다.

정치적인 입장에 덧붙여 그는 여러 소프트웨어 프로젝트로도 널리 알려져 있는데, 가장 유명한 프로젝트는 GNU C 컴파일러인 GCC와 이맥스 편집기다. GCC는 전 세계에서 가장 많은 기종으로 이식된 가장 인기 있는 컴파일러다. 그러나 RMS는 이맥스 편집기로 더 널리 알려져 있다. 이맥스 편집기를 편집기라고 부르는 것은 마치 지구를 양질의 흙덩어리라고 부르는 것과 같을 것이다. 이맥스는 편집기인 동시에 웹 브라우저, 뉴스리더, 메일 소프트웨어, 개인 정보 관리 프로그램, 조판 프로그램, 프로그래밍 편집기, 진수 편집기, 워드프로세서 그리고 비디오 게임으로 사용할 수 있다. 이맥스를 실행한 뒤에 컴퓨터에서 다른 작업을 하기 위해 이맥스를 떠날 필요를 느끼지 않는 프로그래머가 많이 있으며, 이맥스를 사용하게 되면 이것이 단순한 프로그램이 아닌 하나의 종교이며, RMS는 그 종교의 성자라는 사실을 발견할 수 있을 것이다.



マイ클 티만 Michael Tiemann은 시그너스 솔루션즈의 창업자다. 그는 GNU C 컴파일러와 GNU C++ 컴파일러, 그리고 GDB에 대한 작업을 통해 자유 소프트웨어 개발에 기여하기 시작했다. GCC를 SPARC스택과 그 외 몇몇 RISC리스크 아키텍처로 직접 이식했으며, GNU C++ 컴파일러를 만들고 GDB가 C++를 지원하도록 개선하고 이를 SPARC 아키텍처로 이식했다.

기존의 어느 기업도 ‘오픈 소스’ 소프트웨어라는 새로운 형태의 상업 지원 서비스를 확신하지 못하던 1989년에 그는 시그너스 솔루션즈를 공동 창업했다. 마이클 티만은 현재 오픈 소스 소프트웨어와 오픈 소스 사업 모델에 대한 가장 활발한 강연자이자 토론자로 활동하고 있으며, 과거 10년 동안 그랬던 것처럼 향후 10년 동안 흥미와 보상을 얻을 수 있는 기술 사업 솔루션들을 계속 찾고 있다.

1986년 펜실베이니아 대학교 무어 공과대학에서 컴퓨터공학 학사 학위를 취득했으며 1986년부터 1988년까지 텍사스 오스틴의 MCC에서 근무했다. 1988년에는 스탠퍼드 대학원 전자공학과에 입학해 1989년 봄 Ph.D. 과정을 시작했지만 1989년 가을에 시그너스 솔루션즈를 창업하기 위해 학교를 그만두었다.



리누스 토르발스 Linus Torvalds는 누구인가?

물론 그가 리눅스를 창조해 냈다. 이 말은 마치 “엥겔 바트가 마우스를 발명했다.”는 말과 같다. 나는 다음 메일이 오랜 시간이 지난 후 일어날 일에 대한 암시였다고 확신한다.

From: torvalds@klaava.Helsinki.FI (Linus Benedict Torvalds)

Newsgroups: comp.os.minix

Subject: Gcc-1.40 and a posix-question

Message-ID: <1991Jul13.100050.9886@klaava.Helsinki.FI>

Date: 3 Jul 91 10:00:50 GMT

Hello netlanders, Due to a project I'm working on (in minix), I'm interested in the posix standard definition. Could somebody please point me to a (preferably) machine-readable format of the latest posix rules? Ftp-sites would be nice.

안녕하세요. 네트워크 사용자 여러분, 제가 (마닉스에서) 하는 프로젝트 때문에 POSIX 표준 정의가 필요합니다. 누군가 제게 기계어 형식의(이를 더 선호함) 최신 POSIX 규칙이 어디 있는지 알려 주겠습니까? FTP 사이트도 좋습니다.

그러나 이런 일은 일어나지 않았다.

리눅스는 자신의 취미에서 시작한 프로젝트가 7백만에서 1천만 정도의 사용자를 갖는 주요 OS가 되고, 세계에서 가장 큰 소프트웨어 회사와 시장에서 싸우는 가장 큰 경쟁자가 되리라고는 예측하지 못했다.

리눅스가 널리 채택되고 들불처럼 인터넷을 통해 퍼지면서—인터넷 서버의 약 26%가 리눅스로 운영된다(이에 가장 근접한 경쟁자는 마이크로소프트로서 23%)—리눅스 토르발스의 삶은 바뀌었다. 그는 자기가 태어난 핀란드에서 실리콘밸리로 이주했으며 현재 트렌스메타라는 곳에서 일하고 있다. 트렌스메타<sup>01</sup>에서 하는 일에 대해 그는 단지 리눅스와 관련된 일은 아니며 ‘재미있다’라고만 말할 뿐이다.

그는 두 아이와 한 개의 특허권(구성요소의 물리적인 속성에 대한 추론의 실패를 감지하는 마이크로프로세서용 메모리 컨트롤러에게 청혼을 하고 있다)을 가졌고, 핀란드에서 가장 유명한 행사인 대통령의 독립 기념일 무도회에도 초대되었다.

---

01 · 역사주\_트렌스메타는 최근 ‘크루소’라고 부르는 저전력 모바일 칩을 선보였고, 리눅스에서 잘 동작할 것이라고 한다.

자기가 하지 않은 일에 대해 자기가 했다고 명예를 가로채는 일은 그의 성격에 전혀 맞지 않으며, 그는 항상 다른 사람의 도움이 없었다면 리눅스는 오늘날의 리눅스가 될 수 없었을 것이라고 말한다. 데이비드 밀러, 앤런 콕스, 기타 많은 유능한 프로그래머들이 리눅스의 성공에 기여했다. 그들의 도움과 다른 모든 셀 수 없이 많은 사람의 도움이 없었다면, 리눅스는 현재의 위치까지 도약하지 못했을 것이다.



**폴 빅시** Paul Vixie는 빅시 엔터프라이즈 사장으로 bind, inn, dhcp 서버 등을 개발하고 관리했다. 인터넷 소프트웨어 컨소시엄의 설립자이자 대표이기도 하다. 그는 bind의 주된 설계자인데 bind는 DNS를 구현한 가장 인기 있는 소프트웨어다. inn은 인터넷 뉴스서버 패키지며, dhcp는 네트워크 정보를 동적으로 설정하는 데 사용된다.

그는 빅시 크론 Vixie cron의 저자다. 이 프로그램은 리눅스의 기본 크론 데몬으로 사용되며, 다른 운영체제에서도 많이 사용되고 있다. 크론은 예약된 시간에 지정한 프로그램을 자동으로 실행할 수 있도록 해주는 유용한 도구로 매일 새벽 1시에 여러분의 컴퓨터가 자동으로 움직이며 이상한 소음을 내게 하는 주범이다.

그는 『센드메일: 이론과 실제』의 공동 저자다. 그의 회사는 CIX Commercial Internet Exchange를 위한 네트워크를 관리하며, 스팸 발송자에 대한 실시간 블랙홀 리스트를 구성하는 MAPS Mail Abuse Protection System과 함께 스팸 추방 운동에 앞장서고 있다.



래리 월Larry Wall은 rn 뉴스 리더, 어디에서나 볼 수 있는 patch 프로그램, 그리고 펠 언어 등 유닉스에서 가장 널리 사용하는 오픈 소스 프로그램을 만든 사람이다. 그는 또한 Configure 스크립트를 만들어 내는 metaconfig로도 유명하며, 시애틀 태평양 대학에 있을 때 BASIC/PLUS로 만들었던 첫 번째 버전인 워프 우주전쟁 게임으로도 유명하다. 교육 부분에 있어서도 그는 실제 언어학자로서 U.C. 버클리와 UCLA 대학원을 다니기도 했다(이상하게도 그가 U.C. 버클리에 있었을 당시에는 유닉스 개발과는 전혀 관련없는 일을 했다).

래리는 JPL에서 프로그래머로 일했다. 유니시스Unisys에 있을 때는 이벤트 시뮬레이터부터 소프트웨어 개발 방법론까지 다양한 것을 하며 시간을 보냈다. 바로 거기서 Netnews를 조금 고친 버전을 사용하여 1200bps의 암호화된 링크를 통해 양쪽 해안의 설정 관리 시스템을 서로 묶으려고 시도하던 중 펠이 만들어졌다.

현재 래리는 오라일리 미디어에서 일하며 펠에 관한 컨설팅을 한다.



밥 영Bob Young은 항상 오픈 소스 공동체에서 이해할 수 없는 불가사의한 사람, 전설적인 사람으로 불린다. 그는 사업가며, 해커가 아닌 데도 노스캐롤라이나에서 레드햇을 만든, 리눅스계에서 항상 신화적인 인물로 거론된다.

밥은 컴퓨터 리스 산업에서 20년을 보냈고 리눅스 세계로 들어오기 전에 두 회사를 창립했다. 그는 필 휴즈와 SSC가 인수하기 전까지 『리눅스 저널』의 오리지널 출판인이었다. 밥은 마크 유잉Marc Ewing이 이끌던 그 당시 멤버들에게 회사의 금전적인 부분은 걱정하지 않아도 된다고 약속하면서 레드햇에 합류했다. 그는 캡GAP

이나 할리 데이비슨Harley-Davidson 등에서 사용하는 브랜드 규칙을 자유 소프트웨어에 적용했으며, 그 규칙은 대량 소비품, 즉 오픈 소스 소프트웨어를 패키지해서 판매하는 회사에 있어서는 정말로 필요한 것이었다.

레드햇은 원래 자체 제품을 만들어서 직접 마케팅하고 판매하기보다는 상용 OS 회사에 OEM 리눅스 버전을 공급하려고 했다. 그러나 이 회사가 시장에 제 시간에 제품을 내놓지 못해 레드햇이 자사 배포판을 내놓았고 레드햇의 직원들은 그것으로 생계를 꾸려 나갈 수 있었다(스토리는 이랬던 것이다). 레드햇은 최근 벤처캐피털, 넷스케이프, 인텔로부터 자금을 유치했다. 자체 제품을 가지려고 생각지 않았던 레드햇의 성공은 아이러니라고 밖에 할 수 없다.

크리스 디보나Chris DiBona는 1995년 초부터 리눅스를 사용해왔다. 그는 리눅스 공동체에 매우 열심이다. 리눅스 인터내셔널의 웹 마스터로 자원해서 일하며, 리눅스 인터내셔널이 인정하는 자금 코디네이터다. 그는 VA 리서치 리눅스 시스템사의 리눅스 마케팅 디렉터로 일하며, 실리콘밸리 리눅스 사용자 그룹(세계에서 가장 큰 사용자 그룹)의 부회장을 맡고 있다.

리눅스 활동 외에도 저술 작업과 서적 리뷰 등을 비엔나 타임즈, 리눅스 저널, 테크워크, 부트 매거진(현재는 맥시멈 PC), 그리고 수많은 온라인 출판물에 게재하고 있다. 또한 주간 테러리스트 프로파일이라고 해서 약 20,000명의 가입자를 가지고 있는 지역 정치 주간지의 편집장을 2년 동안 하기도 했다.

그의 개인 웹사이트 주소는 <http://www.dibona.com>이며, [chris@dibona.com](mailto:chris@dibona.com)으로 메일을 통해 연락할 수 있다.

**샘 오크만**Sam Ockman은 주문생산 방식의 리눅스 시스템을 전문으로 하는 펭귄컴퓨팅사의 사장이다. 그는 LINC<sup>International Linux Conference and Exposition</sup>의 회장이며, 현재는 리눅스 월드와 합병되었다. 샘은 리눅스 시스템 설치와 설정, 펄 전문가며, U. C. 버클리 화장 학교에서 강의를 하기도 했다. 또한 실리콘밸리와 베이 지역 리눅스 사용자 그룹의 강사이기도 하다.

샘은 유닉스와 펄에 관한 책을 편집했고, 매달 리눅스에 대한 칼럼을 쓴다. 스텐퍼드에서 컴퓨터 시스템 엔지니어링과 정치 과학 학위를 받았다. 샘은 스텐퍼드에 있었을 당시, 연극에서 최고의 배우에게 주는 상 중 하나인 Ram's Head Dorhtea상을 받은 사실에 대해 매우 자랑스럽게 생각한다.

**마크 스톤**Mark Stone은 커널 버전 1.0.8 시절부터 주 운영체제로 리눅스를 사용해왔다. 그는 70년대 후반 첫 번째 대형 프로그램을 작성한 바 있다. 바로 PDP-1170 머신용 알골 컴파일러다. 요즘 그는 컴파일보다는 스크립트에 더 관심을 가지고 있으며, 주로 Tcl을 사용한다.

현재 마크는 오라일리의 오픈 소스 편집자다. 출판계에 입문하기 전에는 철학 교수였으며 로체스터 대학에서 박사 학위를 받았다. 대학에서는 카오스 이론과 과학 철학을 연구했다. 따라서 그가 하는 일의 성격이 그리 크게 바뀌었다고 할 수는 없다.

# 역자 소개

## 옮긴이\_ 송창훈

컴퓨터공학과 법학을 공부했고 오랫동안 GNU 프로젝트에 참여했다. 자유 소프트웨어 운동의 정신이 다른 시대, 다른 사회, 다른 분야에서도 상쾌한 *cheerful* 바람이 될 수 있기를 희망한다. 독자 모두가 프로그래머는 아닐 것이다. 그러나 조금의 유머와 재치, 상냥한 미소와 배려, 그리고 삶의 작은 곳에서도 새로움 *technology*과 흥미 *love*를 찾아간다면 우리 모두 삶의 친구 *life hacker*로 만나게 되리라 생각한다.

## 옮긴이\_ 이기동

금년 2월에 중앙대학교 컴퓨터공학과를 졸업했고 현재 쓰리알 소프트에서 웹메일 시스템 개발을 맡고 있다. 1997년부터 프로그램 세계 리눅스 저널을 번역한 경험이 있다.

## 옮긴이\_ 이만용

서울대학교 자연과학대 지질 과학과를 다녔으며 국내 최초 한글 배포판 ‘알짜 리눅스’ 프로젝트에 참여했다. PC 통신 나우누리 리눅스 동호회 시습으로 활동하기도 했으며, 다수의 중소 기업 서버를 구축했고, SI 프로젝트를 추진했다.

1998년 한국 리눅스 비즈니스(주)를 설립했고, 리눅스 코리아(주)의 기술 이사로 재임했다. 집필한 책으로는 『한글 리눅스 알짜 슬랙웨어 3.1』(정보문화사), 『리눅스 서버 가이드』(정보문화사), 『초보자용 리눅스 프로그래밍』(도서출판 대림), 『한글 알짜 레드햇 5』(정보문화사), 『한글 알짜 레드햇 5.2』(정보문화사)가 있으며, 번역한 책으로는 『러닝 리눅스 Running Linux』(한빛미디어, 1999)가 있다.

## 옮긴이\_ 최준호

최준호는 서울대학교 계산통계학과 통계학을 전공하고 같은 대학의 전산과 학과 석사를 졸업했다. 웹데이터뱅크(주)의 개발팀장을 역임하면서 qLinux 개발을 진행했고, Korea FreeBSD Users Group 운영자이며 GNU Free Translation Project 한국어팀 리더를 맡고 있다. 또한 nh2ps, nhppf 등의 오픈 소스 프로그램과 넷스케이프 한글화 키트등 여러 오픈 소스 프로젝트를 운영하거나 꾸준히 참여했던 경험이 있다.

## 역자 서문

이 책은 리눅스로 대표되는 전 세계적인 오픈 소스 혁명에 더욱 많은 사람이 이해하고 확신하며 동참할 수 있도록 다양한 오픈 소스 소프트웨어(또는 자유 소프트웨어) 리더들의 에세이를 담은 모음집이다. 이 모음집에는 리눅스, 래리 월을 포함한 프로그래머의 기술적인 입장과 리처드 스톤먼, 에릭 레이먼드를 포함한 정신적, 이론적 리더의 주장, 그리고 밥 영, 팀 오라일리를 포함한 오픈 소스 벤처 사업가의 시장 가치의 관점 등이 모든 것이 들어 있어 독자들은 표면적으로는 같다고 느껴지는 오픈 소스 혁명이 사실은 정말로 가지각색인 노력의 총체적인 결과라는 사실을 알게 될 것이며, 그 기반이 매우 튼튼하다는 사실에 놀랄 것이다.

많은 사람은 오픈 소스 소프트웨어가 해커, 아카데미, 시장과는 관련이 없는 곳으로부터 시장을 넘보고 시장을 변화시켜가는 현실적인 힘으로 등장했다는 사실에 놀라고 있다. 사실 이 혁명의 소용돌이 안에서 자기 역할을 열심히 수행하고 있는 개발자와 사업가들도 서로에 대한 존경을 보내고 있을 정도다. 이 책을 들고 읽을 것인가 망설이는 사람들 모두는 언론 매체를 통해 쏟아져 나오는 소식의 실체를 알고 마음의 안정, 미래에 대한 안전한 확신을 하고 싶을지 모른다. 모든 사람은 희망이든 두려움이든 미래에 대하여 알고 싶어하지 않는가? 오픈 소스는 어떨까?

첫 번째 답은 분명하다. 바로 이들이 아니면 누구에게서 오픈 소스 혁명의 실체와 원동력에 대한 대답을 구할 수 있단 말인가? 역자는 이들과 만나 그들의 생생한 목소리로 그들의 꿈, 현실 감각, 자기 업무에 임하는 태도를 느끼고자 3년 전부터 그들이 참석하는 행사에 따라다녔던 것 같다. 그런데 이 에세이집에는 역자가 3년간 돌아다니면서 들었던 것보다 더 많은 이야기가 담겨 있다. 물론 그들의 숨소리를 들을 수는 없지만 그 많은 이야기를 한 곳에서 볼 수 있다는 사실만으로도 충분하지 않은가? 두 번째 답은 불투명하다. 아마도 여러분은 복음서 수준의 확신을 원할

지 모른다. 그러나 이 에세이집은 이미 훨씬 오래전에 지나가서 결과를 모두 알고 있고, 또다시 새로운 변화의 소용돌이 속에서 재평가하는 그런 역사책이 아니다. 이 에세이집은 분명히 말해서 “현재 벌어지고 있는 혁명”의 순간에 쓰고 있는 미완성의 선언문에 가깝다.

역자가 속한 리눅스 비즈니스는 매우 자연스러운 진화의 과정이라고 보고 있는데, 투자자를 비롯하여 외부의 많은 사람은 내게 오픈 소스의 미래가 확고한 것인지, 확신해도 되는지 묻곤 한다. 그럴 때는 이렇게 말하여 그 질문을 교묘히 피하곤 한다. “그런 모든 것이 누구나 믿을 수 있는 현실이라면, 누구나 확신을 가지고 리눅스 비즈니스에 달려들었을 테고, 그럼 모험적인 성공은 더 이상 없는 것 아니겠어요? 진정한 성공은 남들이 성공에 대한 확신을 하지 못하는 영역에서 나올 수 있겠죠.” 그리고는 웃는다. 확신은 여러분의 마음속에 존재하는 것이지 상대방의 말이나 글속에 존재하지 않는다. 하지만 오픈 소스 리더들의 이 소중한 글은 오픈 소스 개발자가 되고 싶은 이들 혹은 오픈 소스 벤처 기업가가 되고자 하는 이들이 자기 확신을 세우는 데 가장 큰 도움을 줄 것이라는 사실을 분명히 말할 수 있다.

번역에 참가하면서 역자가 느꼈던 새로운 감동을 여러분도 느낄 수 있기를 바란다.

May the Open Source be with you!

2000년 5월 26일 역자를 대표하여

리눅스코리아 CTO 이만용

# 감사의 글

어머니와 아버지, 트리시Trish, 데니스Denise와 네il Neil, 그리고 미키Mickey 모두에게 너무 감사드린다. 이렇게 많은 사람에게 조언과 도움을 받으면서 집필한 책은 이번이 처음일 것이다. 나를 도와준 Coffeenet 사람들에게도 심심한 감사를 드리고 싶다. 그리고 물론 이 책을 위해 코딩할 시간을 할애해 준 공헌자들께도 감사드린다. 또한 VA 리눅스 시스템 연구소의 사람들처럼 명석하고 혁신적인 사람들을 본 적이 없다. 내가 글을 쓰는 동안 참고 도와준 그들에게 감사하는 바다.

이 책은 마크 스톤Mark Stone의 지속적인 도움과 혁신으로 완성되었다. 그는 “이 책이 어떻게 쓰였는가에 대해 쓰면 그게 바로 책이지.”라고 말한이 책의 탄생에 있어 진정한 숨은 공로자다. 그 말을 듣고 가능한 최선의 방도가 무엇인가에 대해 확신하게 되었다. 언젠가는 그때를 회상하면서 웃을 것이라고 생각한다. 그렇죠, 마크?

이 책을 쓰기 위한 초창기에 서로 머리를 맞대고 의견을 나눌 때, 결국에는 많은 사람이 오픈 소스로 우리를 이끌어 줄 아이디어를 내주었고 많은 지지를 보내주었다. 폴 크로울리Paul Crowley, 폴 러셀Paul Russell, 코리 살티엘Corey Saltiel, 에드워드 아비스Edward Avis, 제프 리키아Jeff Licquia, 제프 뉴스Jeff Knox, 베基 우드Becky Wood와 <http://slashdot.org>의 촉매 역할을 했던 롭 말다Rob Maldan 등 그들 모두에게 감사한다. 나는 이 책이 여러분들이 그렇게 되기를 원했던 모든 것을 담았기를 바란다.

마지막으로 키보드 앞에 앉아 내 할 일에 몰두해 있을 때 아무 대가도 바라지 않고 내 아파트를 보살펴 준 크리스틴 힐머Christine Hillmer의 혁신이 없었다면 이 일을 도저히 마무리할 수 없었을 거라는 말을 덧붙이고 싶다. 여러분은 내가 희망할 수 있는 모든 것이며, 나는 내가 얼마나 행운아였는지 매일 되새길 것이다.

— 크리스 디보나Chris DiBona

다양한 방면에서 헌신적으로 나를 도와준 조 매퀸Joe McGuckin, 피터 핸드릭슨Peter Hendrickson, 조 슈스터Jo Schuster, 루스 오크만Ruth Ockman, 앤리슨 하인Allison Huynh과 니나 우다드Nina Woodard에게 감사하는 바다. 또한 나와 절친한 해커인 데이비드 밀러David Miller와 피터 앤빈Peter Anvin에게도 고맙다는 말을 전하고 싶다.

— 샘 오크만Sam Ockman

처음에 이 놀라운 아이디어를 제공해준 샘과 크리스에게 감사한다. 나는 샘과 크리스가 참가하려는 생각이 없음을 알고 있었지만 나에게는 그들의 참여가 가치 있는 일이라고 생각되었다. 또한 다른 저자들에게도 감사한다. 그들은 관을 완성할 시간 보다는 아이디어를 더 많이 지닌 창조적인 정신을 가진 사람들이다. 하지만 이 프로젝트의 중요성을 누구보다 잘 아는 그들이었기에 그에 상응하는 상당한 시간을 할애해 주었다.

모든 책 뒤에는 그 책을 성공으로 이끌기 위해 노력하는 수많은 사람이 숨어 있다. 특히 이 책의 출간을 위해 마땅히 감사받아야 할 영웅들이 있다. 트로이 모트Troy Mott, 캐시 가드너Katie Gardner, 타라 맥골드릭Tara McGoldrick, 제인 엘린Jane Ellin, 로버트 로마노Robert Romano, 론 포터Rhon Porter와 낸시 울프 코타리Nancy Wolfe Kotary, 셰릴 Sheryl과マイ크 시에라Mike Sierra, 그리고 에디 프리먼Edie Freeman이 바로 그 영웅들이다. 또한 달리 부르면 ‘아버지를 미치광이로 만든 책 프로젝트’를 위해 애써준 나의 친구들과 가족들에게도 고맙다고 말하고 싶다.

마지막으로 Lytton Coffee Roasting 사의 사람들에게 감사의 말을 전한다.

— 마크 스톤Mark Stone

# 한빛 eBook 리얼타임

한빛 eBook 리얼타임은 IT 개발자를 위한 eBook입니다.

요즘 IT 업계에는 하루가 멀다 하고 수많은 기술이 나타나고 사라져 갑니다. 인터넷을 아무리 뒤져도 조금이나마 정리된 정보를 찾는 것도 쉽지 않습니다. 또한 잘 정리되어 책으로 나오기까지는 오랜 시간이 걸립니다. 어떻게 하면 조금이라도 더 유용한 정보를 빠르게 얻을 수 있을까요? 어떻게 하면 남보다 조금 더 빨리 경험하고 습득한 지식을 공유하고 발전시켜 나갈 수 있을까요? 세상에는 수많은 종이책이 있습니다. 그리고 그 종이책을 그대로 옮긴 전자책도 많습니다. 전자책에는 전자책에 적합한 콘텐츠와 전자책의 특성을 살린 형식이 있다고 생각합니다.

한빛이 지금 생각하고 추구하는, 개발자를 위한 리얼타임 전자책은 이렇습니다.

## 1. eBook Only - 빠르게 변화하는 IT 기술에 대해 핵심적인 정보를 신속하게 제공합니다.

500페이지 가까운 분량의 잘 정리된 도서(종이책)가 아니라, 핵심적인 내용을 빠르게 전달하기 위해 조금은 거칠지만 100페이지 내외의 전자책 전용으로 개발한 서비스입니다. 독자에게는 새로운 정보를 빨리 얻을 수 있는 기회가 되고, 자신이 먼저 경험한 지식과 정보를 책으로 펴내고 싶지만 너무 바빠서 염두를 못 내시는 선배, 전문가, 고수분에게는 보다 쉽게 집필하실 기회가 되리라 생각합니다. 또한 새로운 정보와 지식을 빠르게 전달하기 위해 O'Reilly의 전자책 번역 서비스도 하고 있습니다.

## 2. 무료로 업데이트되는, 전자책 전용 서비스입니다.

종이책으로는 기술의 변화 속도를 따라잡기가 쉽지 않습니다. 책이 일정한 분량 이상으로 집필되고 정리되어 나오는 동안 기술은 이미 변해 있습니다. 전자책으로 출간된 이후에도 버전 업을 통해 중요한 기술적 변화가 있거나, 저자(역자)와 독자가 소통하면서 보완되고 발전된 노하우가 정리되면 구매하신 분께 무료로 업데이트해 드립니다.

### 3. 독자의 편의를 위하여, DRM-Free로 제공합니다.

구매한 전자책을 다양한 IT기기에서 자유롭게 활용하실 수 있도록 DRM-Free PDF 포맷으로 제공합니다. 이는 독자 여러분과 한빛이 생각하고 추구하는 전자책을 만들어 나가기 위해, 독자 여러분이 언제 어디서 어떤 기기를 사용하시더라도 편리하게 전자책을 보실 수 있도록 하기 위함입니다.

### 4. 전자책 환경을 고려한 최적의 형태와 디자인에 담고자 노력했습니다.

종이책을 그대로 옮겨 놓아 가독성이 떨어지고 읽기 힘든 전자책이 아니라, 전자책의 환경에 가능한 최적화하여 쾌적한 경험을 드리고자 합니다. 링크 등의 기능을 적극적으로 이용할 수 있음은 물론이고 글자 크기나 행간, 여백 등을 전자책에 가장 최적화된 형태로 새롭게 디자인하였습니다.

앞으로도 독자 여러분의 충고에 귀 기울이며 지속해서 발전시켜 나가도록 하겠습니다.

지금 보시는 전자책에 소유권한을 표시한 문구가 없거나 타인의 소유권한을 표시한 문구가 있다면 위법하게 사용하고 계실 가능성이 높습니다. 이 경우 저작권법에 의해 불이익을 받으실 수 있습니다.

다양한 기기에 사용할 수 있습니다. 또한 한빛미디어 사이트에서 구입하신 후에는 횟수에 관계없이 다운받으실 수 있습니다.

한빛미디어 전자책은 인쇄, 검색, 복사하여 붙이기가 가능합니다.

전자책은 오탈자 교정이나 내용의 수정보완이 이뤄지면 업데이트 관련 공지를 이메일로 알려드리며, 구매하신 전자책의 수정본은 무료로 내려받으실 수 있습니다.

이런 특별한 권한은 한빛미디어 사이트에서 구입하신 독자에게만 제공되며, 다른 사람에게 양도나 이전되지 않습니다.

# 업데이트

2013. 07. 31

- 복간판 발행

2013. 09. 20

- 저자 소개 및 역자 소개 추가
- 페이지 증가(204쪽 → 218쪽)
- 서문, 3장, 4장, 5장, 6장 미주 부분에 상호 링크 추가(미주 부분 앞을 누르면 해당 본문 페이지로 이동)
- 3장 수정 번역(송창훈 님)

2013. 12. 12

- 페이지 증가(218쪽 → 231쪽)
- 3~6장 수정 번역(송창훈 님)

# 차례

## 서문

1

---

프롤로그.....	1
자유 소프트웨어는 무엇이며 오픈 소스와 어떤 관계를 갖는가?.....	3
오픈 소스 소프트웨어란 무엇인가? .....	4
포스의 어두운 면.....	6
소스를 사용하거라! 루크.....	7
과학적 방법을 통한 기술혁신.....	10
오픈 소스에 대한 위협.....	17
오픈 소스 해커에게 동기 부여하기.....	19
리눅스 벤처와 미래 투자.....	21
과학과 새로운 르네상스.....	23

0 1

## 해커 문학의 짧은 역사

27

---

프롤로그 -진정한 프로그래머.....	27
초기 해커들.....	28
유닉스의 부상.....	32
오랜 시대의 끝.....	34
상용 유닉스 시대.....	36
초기의 공개 유닉스.....	39
웹의 폭발적인 성장.....	40

0 2	<b>버클리 유닉스의 20년</b>	42
	초기 역사.....	42
	초기 배포판.....	45
	VAX 유닉스.....	46
	DARPA의 지원.....	47
	4.2BSD.....	49
	4.3BSD.....	53
	네트워킹, 릴리즈 1.....	55
	4.3BSD - Reno.....	56
	네트워킹, 릴리즈 2.....	57
	소송 .....	60
	4.4BSD.....	62
	4.4BSD - Lite 릴리즈 2.....	63
0 3	<b>인터넷 엔지니어링 태스크포스</b>	65
	IETF의 역사.....	66
	IETF의 구성과 특성.....	67
	IETF 워킹 그룹.....	68
	IETF 문서.....	68
	IETF 표준화 과정.....	69
	공개 표준, 공개 문서 그리고 오픈 소스.....	72

---

최초의 소프트웨어 공유 공동체.....	76
공동체의 붕괴.....	77
피할 수 없는 도덕적 선택.....	79
구속되지 않는다는 관점에서의 자유.....	81
GNU 소프트웨어와 GNU 시스템.....	82
프로젝트의 시작.....	82
첫 번째 단계.....	83
GNU 이맥스.....	84
프로그램은 누구에게나 자유로운가?.....	85
카피레프트와 GNU GPL.....	86
자유 소프트웨어 재단.....	87
자유 소프트웨어 지원.....	88
기술 목표.....	89
기증받은 컴퓨터.....	90
GNU 태스크 리스트.....	90
GNU 라이브러리 GPL.....	91
가려운 곳을 긁는다?.....	92
예기치 않은 개발.....	93
GNU 허드.....	94
알리스.....	94
리눅스와 GNU/리눅스.....	95
미래의 도전들.....	96
비밀 하드웨어.....	96

0 5	비자유 라이브러리.....	97
	소프트웨어 특허.....	99
	자유 문서.....	99
	우리는 자유에 대해 이야기해야만 한다.....	101
	오픈 소스.....	102
	한번 해보자!	103
0 5	<b>시그너스 솔루션즈의 미래: 한 기업가 이야기</b>	<b>113</b>
	초창기의 시그너스.....	123
	GNUPro.....	125
	도전들.....	133
	오픈 소스를 넘는 투자 - eCos.....	136
	미래에 대한 생각과 전망.....	141
0 6	<b>소프트웨어 공학</b>	<b>161</b>
	소프트웨어 공학 공정.....	161
	시장 요구사항 분석.....	162
	시스템 설계.....	162
	상세 설계.....	163
	구현.....	163
	통합.....	164
	현장 테스트.....	165
	사후 지원.....	165

테스트 세부 사항.....	166
코드 커버리지 분석.....	166
회귀 테스트.....	168
오픈 소스 소프트웨어 공학.....	168
시장 요구사항 분석.....	169
시스템 설계.....	170
상세 설계.....	170
구현.....	171
통합.....	172
현장 테스트.....	173
사후 지원.....	173
결론.....	175
 첨단의 리눅스	183
아미가와 모토로라 포트.....	184
マイ크로커널.....	185
알파로부터 이식성까지.....	188
커널 영역과 사용자 영역.....	190
GCC.....	192
커널 모듈.....	193
오늘날의 이식성.....	195
리눅스의 미래.....	196

# 서문

크리스 디보나Chris DiBona, 샘 오크만Sam Ockman, 마크 스톤Mark Stone

이만용 역

## 프롤로그

리눅스 창시자인 리누스 토르발스Linus Torvalds에 의하면 리누스라는 이름은 부모님께서 노벨상 수상자인 리누스 폴링Linus Pauling을 존경했기 때문에 지어졌다고 한다. 폴링은 한 번도 아니고 두 번씩이나 노벨상을 받은, 훈치 않은 과학자다. 우리는 DNA 구조를 발견할 수 있도록 해 준 폴링의 기본적인 역할을 통해 오픈 소스 공동체Open Source Community의 교훈적인 이야기를 발견할 수 있다.

왓슨의 유명한 저서인 ‘이중 나선 구조The Double Helix’에서 살펴볼 수 있듯이 DNA 구조에 대한 실제 발견은 프란시스 크릭Francis Crick과 제임스 왓슨James Watson에 의해 이루어졌다. 왓슨은 그의 저서에서 과학이 실제로 어떻게 이루어지고 있는가에 대하여 정말로 솔직하게 다루고 있으며 탁월함과 통찰력뿐만 아니라 정치적 상황, 경쟁, 행운 또한 강조하고 있다. DNA의 비밀에 대한 탐구는 그 누구보다도 케임브리지에 있는 왓슨 크릭 연구소와 칼 테크에 있는 폴링 연구소 사이를 치열한 경쟁 관계로 만들었다.

왓슨은 저서에서 자신과 크릭이 기존의 미스테리를 해결하고 DNA 나선 구조 모델을 만들었다는 사실을 폴링이 알게 된 경위에 대해 상당히 불쾌하게 생각한다는 어조로 기술하고 있다. 이 이야기의 중심에는 케임브리지와 칼 테크의 두 연구소를 오가며 친분 관계에 있었던 맥스 델브룩Max Delbrück 있다. 결과를 확신하기 전까지

는 비밀을 유지하고자 했던 왓슨과 크릭에게는 안된 이야기지만 텔브룩의 행동은 궁극적으로 과학 자체에 이바지했다. 다음 인용문에서 왓슨은 폴링이 어떻게 그 사실을 알게 되었는지에 대해 설명하고 있다.

먼저 리누스 폴링은 맥스 텔브룩에게 이중 나선 구조에 대한 이야기를 들었다. 나는 보조 사슬에 대한 소식을 전하는 편지 말미에 폴링에게 이야기하지 말아 달라고 텔브룩에게 요청했다. 아직은 무언가 잘못되지 않을까 하는 미심쩍은 상태였기 때문에 우리의 입장장을 정리하기 위한 며칠의 여유를 갖기 위해서였다. 그때까지는 폴링이 수소가 결합한 기본 쌍에 대해 생각하지 않기를 바랬다. 하지만 내 요청은 무시당했다. 텔브룩은 그의 연구소에 있는 모든 사람에게 이야기하기를 원했다. 몇 시간 만에 그 이야기는 그의 생물학 연구소로부터 리누스 밑에서 일하는 다른 친구들의 귀로 들어가게 되었다. 또한 텔브룩은 나에게 전해 들은 소식을 폴링에게 전해 줄 것을 약속했었다. 그리고 무엇보다도 중요한 사실이 있다. 텔브룩은 과학적인 영역에서 모든 형태의 비밀을 거부했으며 폴링의 연구가 중단되기를 바라지 않았던 것이다.

비밀에 부쳐야 한다는 사실 때문에 왓슨이 불편했던 것은 분명하다. 참고로 이 책 전반에 걸쳐 전개되는 신랄한 주제 중 하나는 참여자들로 하여금 자신이 아는 것을 감추도록 만드는 경쟁이 있다는 점이다. 그는 아무리 작은 것일지라도 결국 비밀은 과학의 진보를 느리게 한다는 것을 주지하고 있었던 것이다.

과학은 결국 오픈 소스 엔터프라이즈라고 말할 수 있다. 즉, 과학적인 방법이란 발견과 증명의 과정에서 비롯한다. 과학적인 결과를 입증하기 위해서는 그 결과를 재현할 수 있어야 하며, 재현은 소스source가 공유되지 않는 한 불가능하다. 그 소스란 바로 가설, 실험조건, 결과다. 발견의 과정 다음에는 많은 길이 놓여 있으며 때로는 서로 고립된 채로 과학적 발견이 이루어지기도 한다. 하지만 발견의 과정은 정보 공유를 통해 이루어져야 한다. 즉, 혼자서는 갈 수 없는 곳까지 다른 과학자들이 전진할 수 있도록 하는 것이며, 다른 사람의 아이디어에 수분(受粉) 됨으로써 고립된 상태로는 불가능한 새로운 무언가가 자라날 수 있도록 하는 것이다.

## 자유 소프트웨어는 무엇이며 오픈 소스와 어떤 관계를 갖는가?

1984년 MIT 인공 지능 연구원이었던 리처드 스톤먼Richard Stallman은 GNU 프로젝트를 시작하였다. GNU 프로젝트의 목적은 간단히 말해 누구도 소프트웨어에 비용을 지불하지 않도록 하는 것이었다. 스톤먼은 실행 프로그램을 구성하는 지식(컴퓨터 산업에서는 소스 코드라고 부르는)이 공개되어야 한다고 생각했기 때문에 GNU 프로젝트를 시작하였다. 그렇지 않으면 소수만이 컴퓨팅을 지배하는 폐해를 낳을 것이다.

독점적인 상업용 소프트웨어 업체는 과학적 지식이 철저히 보호되고 비밀이 유지되어야 한다고 주장하지만 리처드 스톤먼은 널리 공유하고 배포해야 하는 것으로 본다. GNU 프로젝트와 자유 소프트웨어 재단Free Software Foundation, FSF—GNU 프로젝트의 우산 조직이다—의 기본 신조는 소스 코드는 컴퓨터 과학을 발전시키는데 기본이며, 변화가 계속되기 위해 정말로 필요한 것은 바로 자유롭게 사용할 수 있는 소스 코드라는 것이다.

스톤먼은 세상이 자유 소프트웨어에 대하여 어떠한 반응을 보일 것인가를 염려하였다. 과학적 지식은 보통 공공 영역public domain에 속하며 학문 분야의 출판 쪽에서 이 일을 담당한다. 그러나 소프트웨어의 경우는 소스 코드를 그냥 공공 영역에 내놓으면 기업에서 자기 자신의 수익을 위해 해당 코드를 흡수하게 될 것이 분명하다. 이러한 위협에 대해 스톤먼은 GPL이라고 알려진 GNU 일반 공중 이용 허락 GNU General Public License로 대처하였다.

GPL이 기본적으로 말하는 바는 다음과 같다. 여러분은 마음대로 GPL 라이선스 소프트웨어를 복사하거나 배포할 수 있다. 그러나 소프트웨어 그 자체에 비용을 매기거나 다른 라이선스를 적용해서 다른 사람이 여러분과 같은 권리를 갖지 못하도록 막아서는 안 된다. GPL은 또한 GPL 라이선스 작업에서 유추된 작업도 마찬가지로 GPL 라이선스를 가져야 한다고 요구한다.

스톨먼을 비롯한 이 책의 저자들이 자유 소프트웨어에 대하여 말할 때는 사실 언론의 자유<sup>free speech</sup>에 대하여 말하는 것이다. 영어에서는 이 차이점을 제대로 구분하지 못하지만 “Free as in speech, not as in beer”에서처럼 무료<sup>gratis</sup>와 자유 liberty는 구분되어야 한다. 이러한 급진적인 메시지(맥주 부분이 아니라 자유 부분의) 때문에 많은 소프트웨어 회사는 자유 소프트웨어를 거부하고 있다. 회사들이 란 지식의 체계에 이바지하기보다는 결국 돈을 버는 비즈니스 영역에 있기 때문이다. 추측하건대 스톤먼은 컴퓨터 산업과 컴퓨터 과학 사이의 분열을 당연한 것으로 받아들이며 심지어는 원한다고 할 수 있다.

## 오픈 소스 소프트웨어란 무엇인가?

1997년 봄, 자유 소프트웨어 공동체의 리더 그룹들이 캘리포니아에서 회합을 했다. 이 그룹에는 에릭 레이먼드 Eric Raymond, 팀 오라일리 Tim O'Reilly, VA 리서치사 사장인 래리 오거스틴 Larry Augustin이 포함되어 있었다. 그들의 관심사는 기존 자유 소프트웨어라는 개념에 대해 거부감을 가진 사람들에게 자유 소프트웨어를 둘러싼 아이디어들을 진홍시킬 방법을 찾는 것이었다. 즉, 자유 소프트웨어 재단의 반(反) 비즈니스적인 메시지가 대체로 자유 소프트웨어의 힘을 진정하게 평가할 수 없게 한다고 걱정하였다.

에릭 레이먼드의 끈질긴 주장으로, 회합에 모인 사람들은 시장 점유뿐만 아니라 대중들의 인식까지도 점유할 수 있는 마케팅 캠페인이 부족했다는 점에 동의했다. 회의 도중 진홍시키고자 하는 소프트웨어를 나타내는 새로운 용어인 오픈 소스<sup>Open Source</sup>가 나왔다. 그리고 오픈 소스라고 말할 수 있는 소프트웨어에 대한 가이드라인을 만들었다.

브루스 페렌스 Bruce Perens는 오픈 소스 정의에 대하여 많은 기초 작업을 해두었다. GNU 프로젝트의 커다란 목표 중 하나는 GNU 소프트웨어를 실행할 수 있는 플랫폼으로써 자유롭게 사용할 수 있는 운영체제를 만드는 것이다. 소프트웨어 부트

과정과 관련하여 리눅스가 그 플랫폼이 되었으며, 이 리눅스는 GNU 도구의 도움을 받아 만들어졌다. 페런스는 GNU 정신을 고수하는 소프트웨어만을 포함한 리눅스 배포판인 데비안Debian 프로젝트를 지휘했었다. ‘데비안 사회 계약Debian Social Contract’이라는 문서에는 이 점을 명시한 바 있다. 오픈 소스 정의는 바로 이 ‘데비안 사회 계약’에서 직접 파생되었으므로 여러 면에서 GNU 정신에 속한다.

오픈 소스 정의는 GPL보다 더 많은 자유를 허용한다. 특히 독점 소프트웨어와 오픈 소스 소프트웨어를 병합하는 상황에서 더 많은 비순수성promiscuity을 허용한다.

결과적으로 오픈 소스 라이선스는 보상이나 크레딧credit 없이도 오픈 소스 소프트웨어를 사용하고 재배포하는 것을 허용한다. 예를 들어 여러분은 넷스케이프사에 통보하지 않은 채 넷스케이프 브라우저 소스 코드와 다른 (독점일 수 있는) 프로그램을 함께 만들어 판매할 수 있다. 왜 넷스케이프사가 이러한 상황을 원하는가? 많은 이유가 있겠지만 그중 가장 확실한 것은 그들이 상업적으로 제공하고 있는 제품 또는 서비스와 매우 잘 작동하는 클라이언트 코드의 시장 점유율을 높여 주기 때문이다. 이러한 소스 코드의 공개는 플랫폼을 만들기 위한 가장 좋은 방법이며, 동시에 넷스케이프사에 있는 사람들이 GPL을 사용하지 않은 이유 중의 하나다.

다음 내용은 공동체에 있어 쉽게 간과할 수 없었던 이슈를 설명한다. 1998년 후반, 리눅스 공동체를 위협하여 분열을 초래할 수 있는 중요한 분쟁이 있었다. 이 분쟁은 각자 객체 지향의 데스크톱 인터페이스를 만들려고 시도한 GNOME과 KDE라는 소프트웨어 시스템에 의해 발생했다. KDE는 트롤 테크놀러지사가 소스 코드를 독점하지만 매우 안정적이고 성숙한 상태의 Qt 라이브러리를 사용하기로, GNOME 개발자들은 Qt처럼 성숙하지는 않았어도 완전히 자유로운 라이브러리인 GTK+ 라이브러리를 사용하기로 하였다.

예전에 트롤 테크놀러지사는 GPL을 사용하든가 아니면 자신의 독점 소유 위치를 지키든가 둘 중 하나를 선택해야 했다. 그렇게 하지 않으면 GNOME과 KDE 사이

의 분열은 계속되었을 것이다. 하지만 오픈 소스 덕분에 트롤 테크놀러지사는 오픈 소스 정의에 부합하면서도 자신이 원하는 대로 제어권을 가질 수 있는 내용으로 자신의 기존 라이선스를 변경할 수 있었다. 덕분에 리눅스 공동체 사이에 존재했던 분열 양상이 점차 사라져 가고 있다.

## 포스<sub>01</sub>의 어두운 면

그 당시에는 인식하지 못했겠지만 왓슨은 생물과학 분야에 있어 새로운 시대의 문턱에 서 있었다. 이중 나선 구조를 발견했을 당시 생물학과 화학 분야는 본질적으로 수공업과 같은 실용적인 기술이라고 할 수 있었다. 또한 그 분야의 연구는 주로 학문 연구소의 후원 아래 작은 그룹으로 일하는 소수에 의해 진행되었다. 그러나 변화의 씨앗은 이미 심어진 상태였다. 소아마비 백신과 같은 몇 가지 의학 분야의 놀라운 진전과 폐니실린의 발견으로 인해 생물과학이 하나의 산업이 되려고 하는 순간이었기 때문이다.

오늘날 유기화학, 분자 생물학, 기초 의학 연구는 소규모 개인 연구원들의 몫이 아니라 하나의 산업이다. 물론 학문 분야의 연구도 진행되지만 대부분의 연구원과 연구비는 제약 산업으로부터 지원받는다. 과학과 산업의 동맹은 근본적으로 불편한 관계일 수밖에 없다. 제약 회사는 학문 분야에선 꿈도 꿀 수 없는 거대한 자금을 지원하지만 기득권을 가지고 연구 자금을 지원하려고 한다. 생각해보라. 제약 회사가 질병에 대한 치료 요법 연구와 투약 요법 연구 중 어느 연구에 자금을 지원할 것으로 생각하는가?

컴퓨터 과학도 마찬가지로 산업과 불편한 동맹 관계에 있다. 과거엔 새로운 아이디어가 주로 학문 분야의 컴퓨터 과학자로부터 나왔다. 하지만 지금은 컴퓨터 산업이 세상의 혁신을 주도한다. 오픈 소스 프로그래머 대부분도 여전히 전 세계적으로 컴퓨터 과학 대학생 또는 대학원생이지만 점점 더 많은 오픈 소스 프로그래머가 학문 분야가 아닌 산업체에서 일하기 시작하고 있다.

산업은 몇 가지 놀라운 기술 혁명을 일으켰다. 예를 들어 이더넷, 마우스, 그래픽 사용자 인터페이스 GUI 모두 제록스 Xerox PARC에서 나왔다. 그러나 컴퓨터 산업에 불길한 징조가 없는 것은 아니다. 레드몬드 Redmond(미국 마이크로소프트사가 위치한 곳) 밖의 누구도 지금의 마이크로소프트사처럼 컴퓨터 데스크톱이 어떤 형태여야 하며 어떤 것을 포함해야 하는지를 독점하는 상황에 대해 좋다고 생각하지 않는다.

산업은 기술 혁신에 부정적인 영향을 미칠 수 있다. 그래픽 이미지 처리 프로그램 GIMP은 0.9 버전의 불완전한 베타 상태에서 1년 동안이나 개발이 멈추어져 있었다. 프로그램을 만든 베클리의 두 학생이 졸업하면서 산업체로 직장을 얻으면서 그들의 기술 혁신을 제쳐 놓았기 때문이다.

## 소스를 사용하거라! 루크<sup>02</sup>

오픈 소스란 위로부터 결정된 아이디어가 아니다. 오픈 소스 운동은 진정한 풀뿌리 혁명이다. 에릭 레이먼드와 브루스 페런스와 같은 에반젤리스트가 자유 소프트웨어와 관련된 언어를 바꾸는 데 성공했다고는 하지만, 그러한 변화는 조건이 제대로 맞지 않았다면 불가능한 일이었다. 이러한 성공의 배경에는 현재 산업체에서 일하는 GNU의 영향 아래 컴퓨터 과학을 배운 학생 세대가 있었다. 이들은 수년간 소리 없이 산업의 뒷문으로 자유 소프트웨어를 들여다 놓았다. 타인을 생각하는 마음에 서라기보다는 모두 그들의 작업에 더 좋은 코드를 사용하기 위해서였다.

혁명가들은 산업의 현장에 있다. 그들은 네트워크 엔지니어, 시스템 관리자, 프로그래머로서 교육을 받는 동안 오픈 소스 소프트웨어를 통해 성장했다. 그리고 전문가가 되기 위해 오픈 소스 소프트웨어를 사용하기를 원한다. 자유 소프트웨어는 종종 무의식적으로 많은 회사의 중요 부분이 되지만 일부 경우는 의도적으로 채택하기도 한다. 오픈 소스는 전성기를 맞이하였고, 이제는 오픈 소스 비즈니스 모델이라는 것도 존재한다.

밥 영 Bob Young의 회사인 레드햇 Red Hat 소프트웨어사는 자사의 핵심 제품인 레드햇 리눅스를 무료로 배포함으로써 번창하고 있다. 자유 소프트웨어를 전달하는 효과적인 방법의 하나는 홀륭한 매뉴얼과 함께 완전한 기능을 갖춘 배포판으로 패키지 하는 것이다. 영은 사람들 대부분이 리눅스 배포판을 구성하는 모든 요소를 다운로드하는 것을 귀찮아한다는 성향을 이용한다. 즉, 편리함을 판다고 할 수 있다.

물론 영만이 이런 사업을 하는 것은 아니다. 그런데 왜 레드햇이 미국 시장을 지배하고 있는가? 왜 SuSE 리눅스가 유럽 시장을 장악하고 있는가? 오픈 소스 소프트웨어는 상품 시장이다. 상품 시장에서는 소비자가 믿을 수 있는 브랜드가 중요하다. 레드햇사의 힘은 브랜드 관리에서 나온 것이다. 지속적인 마케팅, 공동체로의 접근을 통하여 누군가 자기의 친구로부터 어떤 배포판을 사용하면 좋을지 질문을 받을 때마다 레드햇 제품을 추천하도록 만들었다. SuSE도 마찬가지 경우다. 즉, 이 두 회사는 처음으로 브랜드 관리를 진지하게 받아들였기 때문에 각각의 시장을 확보할 수 있었다.

공동체 지원은 필수다. 레드햇, SuSE, 그리고 리눅스 영역의 다른 회사들은 재투자하지 않고도 리눅스로부터 이익을 남기기 위해서는 두 가지 난관을 넘어야 한다는 것을 인지하고 있다. 이 난관의 첫 번째는 사람들이 이런 회사가 오픈 소스 리눅스에 무임승차했다고 생각하여 다른 경쟁자를 권장할 것이라는 데 있고, 두 번째로는 다른 경쟁자와 자신을 차별화해야 한다는 데 있다. 칩바이트 Cheap Bytes와 리눅스 센트럴 Linux Central 같은 회사는 1달러 수준의 저가 배포판 CD를 판매할 뿐이다. 따라서 레드햇이 이러한 싸구려 배포업자보다 더 큰 가치를 제공한다고 공동체에 인식시키려면 무엇인가 되돌려 주어야 한다. 오픈 소스 모델이라는 놀라운 아이러니 속에서 레드햇은 가치를 제공하기 위해 새로운 코드 개발을 지원하고 코드를 대부분 오픈 소스 형식으로 공동체에 되돌려줌으로써 배포판을 49.95달러에 판매할 수 있었다.

이런 브랜드 관리는 오픈 소스에는 새로운 것이다. 하지만 단순히 좋은 서비스를 제공한다는 예전 모델도 오랫동안 오픈 소스 비즈니스 모델의 일부가 되어 왔다. 마이클 티만Michael Tiemann의 경우 세상에서 가장 좋은 컴파일러인 GCC를 무료로 구할 수 있더라도 회사들은 GCC에 대한 지원과 확장 기능에 대하여 비용을 지불할 용의가 있다는 아이디어를 갖고 시그너스Cygnus의 창립을 도왔다.

“자유 소프트웨어를 구입할 수 있게 만든다(Making free software affordable).”

공동 창립자인 존 길모어John Gilmore가 시그너스에 대하여 말한 위 표현이 모든 것을 말해주는 듯하다.

사실 제품을 무료로 제공하고 지원을 판매하는 모델은 현재 오픈 소스 세계에서 빠르게 퍼져 나가고 있다. VA 리서치사는 1993년 후반부터 고품질 리눅스 시스템을 만들고 판매해왔다. 펭귄 컴퓨팅Penguin Computing사도 유사한 제품과 서비스를 제공하며 리눅스 케어Linux Care사는 온갖 종류의 리눅스 지원 서비스를 하고 있다. 센드메일을 만든 에릭 올맨Eric Allman은 시장 점유율 80%를 차지한 메일 서버 소프트웨어에 대한 서비스와 확장 기능을 제공하기 위해 센드메일사를 창립하였다. 특히 센드메일은 시장에 두 가지 방향으로 접근한다는 점에서 흥미 있는 사례다. 센드메일사는 독점 지위에 있는 센드메일 프로 제품과 센드메일 프로의 개발 사이클보다 1년 뒤처진 자유 소프트웨어 센드메일을 가지고 있다.

같은 선상에서 빅시 엔터프라이즈Vixie Enterprise사의 사장이며 이 책에 글을 쓴 저자이기도 한 폴 빅시Paul Vixie는 BIND라는 프로그램을 통하여 실제적인 독점 지위를 누리고 있다. 잘 드러나지 않는 이 프로그램은 여러분이 메일을 보내거나 웹사이트에 가거나, FTP를 통해 파일을 다운로드할 때마다 사용되는 프로그램이다. BIND는 “[www.dibona.com](http://www.dibona.com)” 같은 주소를 실제 IP 주소([www.dibona.com](http://www.dibona.com) 경우 209.81.8.245)로 변환하는 역할을 하는 프로그램이다. 빅시는 널리 퍼져 있는 BIND 프로그램에 대하여 밀려드는 컨설팅을 즐기고 있다.

## 과학적 방법을 통한 기술혁신

오늘날 오픈 소스 운동의 가장 놀라운 성과는 레드햇이나 센드메일과 같은 회사의 성공이 아니다. 더욱 흥미로운 것은 IBM, 오라클과 같은 컴퓨터 산업의 메이저급 회사가 오픈 소스를 비즈니스 기회로 보고 관심을 기울인다는 사실이다. 그렇다면 과연 그들이 오픈 소스에서 찾는 것은 무엇일까? 바로 기술혁신*Innovation*이다.

과학은 결국 오픈 소스 엔터프라이즈며 과학적 방법이란 발견과 입증의 과정에 의존한다. 그리고 과학적인 결과를 증명하기 위해서는 반복 실험이 가능해야 한다. 또한 소스가 공유되지 않으면 반복 실험은 불가능하며 소스란 바로 가설, 실험조건, 결과다. 발견 과정 다음에는 많은 길이 뒤따른다. 때로는 과학적 발견이 개별적으로 고립되어 이루어지기도 한다. 하지만 발견의 과정은 결국 정보 공유를 통해 제공되어야 한다. 이는 혼자서는 할 수 없는 곳까지 다른 과학자들이 전진할 수 있도록 한다. 다른 사람의 아이디어를 응용함으로써 혼자서는 불가능한 새로운 것이 자라날 수 있게 되는 것이다.

과학자들이 결과의 반복을 말한다면 오픈 소스 프로그래머는 디버깅을 말한다. 과학자들이 발견에 대하여 말할 때 오픈 소스 프로그래머는 창조를 말한다. 컴퓨터 산업의 심장부에는 컴퓨터 과학이 있기 때문에 결국 오픈 소스 운동은 과학적 방법의 확장이라고 할 수 있다. 컴파일러의 발명자인 그레이스 하퍼Grace Hopper가 60년 대에 말한 것을 되새겨보자.

내게 있어 프로그래밍은 중요한 실용적 기술 이상의 것이다. 프로그래밍은 또한 지식 기반에 대한 거대한 책임을 지는 것이다.

하지만 컴퓨터 과학은 근본적으로 다른 과학과 다르다. 컴퓨터 과학은 다른 동료들이 결과를 반복할 수 있게 하는 수단을 딱 하나 가지고 있다. 바로 소스 코드의 공유다. 누군가에게 프로그램의 유효성을 입증하려면 그들에게 프로그램을 컴파일하고 실행할 수단을 제공해야 한다.

반복 실험은 과학의 결과를 튼튼하게 만들어준다. 한 명의 과학자가 모든 가능한 실험 조건과 가설의 모든 면을 충분히 실험할 수 있는 환경을 가질 수는 없다. 과학자는 동료들과 가설과 결과를 공유함으로써 한 사람의 눈으로 놓친 것을 다른 많은 눈이 볼 수 있도록 한다. 오픈 소스 개발 모델에서 이 원리는 다음과 같이 표현할 수 있다.

“많은 사람이 볼수록 모든 버그는 사소한 것이다.”

소스 코드를 공유함으로써 오픈 소스 개발자들은 소프트웨어를 더욱 튼튼하게 만든다. 개개인의 프로그래머는 해낼 수 없는 다양한 상황에서 프로그램을 사용하고 테스트하며 발견하기 힘든 버그를 밝혀낸다. 소스 코드가 제공되기 때문에 개발과정 밖에 있는 누군가가 버그를 발견할 뿐만 아니라 종종 고치기도 한다.

과학의 결과를 공개적으로 공유하면 발견을 촉진한다. 즉, 과학의 방법은 다른 사람으로 하여금 유사한 프로젝트를 진행한다는 사실을 알게 해줌으로써 중복 노력을 최소화시킨다. 한 과학자가 프로젝트를 그만두었다고 해서 과학의 진전이 멈춰지지는 않는다. 결과가 가치 있는 것이라면 다른 과학자가 뒤따라 연구할 것이다. 마찬가지로 오픈 소스 개발 모델에서 소스 코드 공유는 창조성을 촉진한다. 서로 보완해줄 수 있는 프로젝트를 수행하는 프로그래머들은 서로의 결과를 향상해주거나 자원을 결합하여 하나의 프로젝트로 만들 수 있다. 혹은 한 프로젝트가 다른 프로젝트에 대한 영감을 줄 수도 있다. 어느 한 프로그래머가 빠진다고 해서 가치 있는 프로젝트가 방치되지는 않는다. 소스 코드를 구할 수 있다면 다른 프로그래머가 참여하여 프로젝트를 이끌어 갈 수 있다. GIMP는 1년간 개발이 멈춰 있었지만 결국에는 개발이 계속되었고, 오늘날 오픈 소스 개발자가 새로운 영역 바로 엔드 유저<sup>end-user</sup> 애플리케이션에서 할 수 있는 것에 대하여 언급할 때 자랑스럽게 GIMP를 지목한다.

미국의 500대 기업은 기술 혁신을 위해 이 모델을 활용하길 원한다. IBM은 아파치를 MIS 부서에 통합하고 관리하기 위해 기꺼이 예산을 따로 편성하였다. 이는 IBM에게 있어서는 성공적이었다. 즉, 놀라울 만큼 안정적인 플랫폼을 설치함으로써 플랫폼 지원 비용을 줄이고, 소비자를 진정으로 도울 수 있는 서비스를 제공할 수 있게 되었다. 또한 IBM 엔지니어는 아파치 팀의 다른 개발자와 아이디어를 서로 교환할 수 있게 되었다.

넷스케이프사가 자사의 브라우저를 오픈 소스로 한 배경이 바로 여기에 있다. 그들 목표의 일부는 시장 점유율을 안정화하거나 증가하는 것에 있었지만 더욱 중요한 목표는 개별적인 개발자가 공동체 기술 혁신을 이끌 수 있을 것이라는 기대에 있었다. 이러한 목표는 그들이 더욱 우월한 제품을 만들도록 도왔다.

IBM은 재빨리 AS400과 같은 서버 플랫폼에 아파치와 같은 소프트웨어 기술을 탄탄하게 결합했다. 그리고 RS6000 생산 라인은 계약을 성사하거나 좀 더 많은 IBM 하드웨어 판매를 도울 수 있게 되었다. IBM은 다음 단계까지 밀고 나가 자사의 인기 있는 DB2 데이터베이스를 리눅스 운영체제에 이식하였다. 많은 사람은 오라클사의 리눅스용 오라클 8 발표에 대한 대응이라고 보고 있지만 IBM은 공동체 안에서의 역할을 진지하게 받아들였고, 자사의 자원을 오픈 소스 목적에 투입하였다. 결과적으로 아파치를 AS400 플랫폼에 이식하여 IBM은 자사의 인기 있는 메인프레임을 차지했고, 오직 IBM만이 할 수 있는 방식으로 오픈 기술을 인정하였다.

Coleman(열전자 분야의), SAIC, BDM, 그리고 IBM과 같은 회사가 연방 정부와 기업에 입찰을 진행할 때 어떤 일이 벌어질지 지켜보는 일은 재미있을 것이다. N T 나 솔라리스를 가지고 1, 000개의 소프트웨어를 설치하는 비용과 리눅스를 가지고 1, 000개를 설치하는 비용을 비교해보자. 백만 달러의 1/4 이상 낮은 가격으로 입찰가를 떨어뜨릴 수 있다면 다른 입찰자보다 훨씬 효과적으로 경쟁할 수 있을 것이다. 계약을 따내기 위해 보통 1~2% 정도의 이익을 포기하곤 하는 CSC와 같

은 회사는 리눅스와 같은 기술을 어떻게 활용할 수 있는지 알려고 노력할 것이다. IBM, 오라클, 넷스케이프사와 같은 회사들이 자신의 비즈니스 모델에 오픈 소스를 통합했지만, 여전히 많은 전통적인 소프트웨어 회사는 오로지 독점 솔루션에만 집착하고 있다. 그들은 위험을 각오하고 있는 것이다.

웹 서버 영역에 있어 마이크로소프트사가 오픈 소스 현상을 전적으로 부인하는 것은 우스운 일이다. 넷크래프트 조사(<http://www.netcraft.com/survey>)에 의하면 아파치 웹 서버는 이 글을 쓰는 지금(2000년) 웹 서버 시장의 50% 이상을 차지하고 있다. 마이크로소프트사의 인터넷 인포메이션 서버 IIS 광고를 보면 자사 제품이 웹 서버 시장의 두 배 이상, 즉 상용 서버 시장의 절반 이상을 장악했다고 말하는 것을 볼 수 있다. 넷스케이프사나 로터스 같은 회사와 비교하면 마이크로소프트는 시장 점유에 있어 확실한 우위를 점하고 있다. 그러나 전체 서버 시장에서 아파치의 53%와 비교하면 마이크로소프트가 주장하는 20%의 ‘확실한 우위’란 보잘것 없어 보인다.

하지만 아이러니는 심화되고 있다. QUESO와 WTF에 의한 조사 자료에 의하면 웹의 29%가 리눅스 기반의 서버로 운영되고 있다. QUESO는 TCP/IP 패킷을 보내서 서버가 어떻게 반응하는가 분석하여 운영체제가 어떤 것인지 파악하는 도구다. 서버의 인식 태그를 분석하는 넷크래프트 질의 엔진과 함께 사용하면 웹 서버 시장의 OS와 서버 타입을 파악할 수 있다.

사실 독점 소프트웨어 업체들은 이미 알게 모르게 타격을 입었다. 리눅스와 FreeBSD는 PC 하드웨어에 독점적인 유닉스를 판매할 기회를 박탈하였다. 그런 회사 중의 하나인 코히런트 Coherent는 이미 파산하였다. 산타크루즈 오퍼레이션 SantaCruz Operation, SCO은 2년 사이에 유닉스 업체의 선두에서 그 뒷전으로 밀려났다. SCO와 같은 회사는 생존의 방법을 찾을지 모르나 주력 제품인 SCO 유닉스가 오픈 소스의 또 다른 희생자가 되지 않을까?

썬Sun은 리눅스 스팩SPARC 이식을 돋기 위해 하드웨어와 자원을 기증하고 존 야우스터하우트John Ousterhout의 Tcl 스크립트 언어 개발을 지원하는 등, 수년간 여러모로 오픈 소스 개발을 지원해왔다. 커크 맥퀴식Kirk McKusick이 말하듯 베클리 기반의 자유 소프트웨어에서 자라난 이 회사가 오픈 소스 현상의 중요성에 대해 인식하려고 노력하는 것은 아이러니다.

다음은 SCO와 썬Sun을 비교한 내용이다.

썬은 그들의 OS와 하드웨어를 지원하고 서비스를 제공하는 것으로 수익 대부분을 창출 한다. 썬의 제품 라인은 PC와 경쟁적인 가격인 데스크톱 워크스테이션부터 메인프레임 영역과 경쟁하는 대형 엔터프라이즈급 서버 클러스터까지다. 하드웨어 영역에서의 이익은 특화되고 사용자화한 E, 그리고 A 시리즈의 서버에 대한 서비스, 판매, 지원으로부터 나오는 것이지 로우엔드 울트라 시리즈에서 나오는 것은 아니다. 썬은 지원, 교육, 컨설팅 서비스 부문에서 수익의 50% 이상을 벌어들인다고 평가받고 있다.

다른 한편으로 SCO는 SCO 유닉스 운영체제, 그리고 컴파일러나 서버와 같은 프로그램, SCO 제품에 대한 훈련과 교육으로 돈을 벌고 있다. SCO는 훌륭하게 조직화를 하고 있지만 마치 하나의 농작물만 다루는 농가가 한 번의 병충해로 큰 피해를 입을 수 있는 것처럼 위기에 처해 있다.

썬Sun은 리눅스 개발을 자사의 로우엔드 제품에 대한 단순한 위협 그 이상으로 보고 있다. 썬의 전략은 리눅스가 썬 하드웨어에서 실행되게 함으로써 사람들이 썬 하드웨어에서 리눅스를 선택할 수 있게 하는 것이다. 이렇게 함으로써 여전히 썬 하드웨어를 지원할 수 있다는 자체가 흥미로운 일이다. 앞으로 썬이 로우 엔드 머신의 리눅스용 소프트웨어를 지원한다고 해도 놀랄 것 없다.

여러 면에서 썬은 작은 초석을 쌓아가고 있다. 사실 썬 관리자를 지금 불러 그에게 썬 박스를 새롭게 구입하고 처음 하는 일이 무엇이냐고 물으면 그는 “GNU 도구와 컴파일러를 다운로드하고 내가 좋아하는 셸을 설치한다.”라고 말할 것이다. 썬은

이러한 메시지를 고객에게 들을 것이며 고객을 포용하는 수단으로 서비스하게 될 것이다. 하지만 썬은 반대로 고객이 그들을 위해 제공할 수 있는 서비스, 즉 소스 코드 발표를 통한 상호 아이디어 교환으로 이를 수 있는 기술 혁신을 이해할 때까지 불리한 처지에 있을 수밖에 없다.

한편 SCO는 유연성에 있어 썬에게 뒤떨어진 모델을 가지고 있다. SCO의 가격 정책은 OS를 먼저 팔고, 이미 리눅스 사용자들이 당연시하는 컴파일러나 텍스트 프로세싱 언어에 대한 부가 비용을 청구하는 것이다. 이 모델은 안정적인 자유 OS와의 경쟁 속에서는 유지될 수 없다. 폭넓은 하드웨어에 가치를 덧붙이는 썬과 달리 SCO는 자신의 이익을 결합할 하드웨어를 갖고 있지 않다. 그들의 OS란 기본적으로 모든 이가 가진 것이며, SCO의 경우 그렇게 훌륭하지도 않다. SCO는 어떻게 할 것인가?

지금까지 그들의 대응은 현명하지 않았다. 1998년 초, SCO는 상당히 광범위한 메일링 리스트를 통해 리눅스나 FreeBSD와 같은 오픈 유닉스가 안정적이지 않으며 비전문가적이라고 혹평하면서 SCO 기반 OS를 할인 가격으로 제공하겠다는 메일을 보낸 적이 있었다. 이들은 이러한 행동으로 인해 많은 사람으로부터 경멸을 당했으며 심각한 후퇴를 해야 했다. 결국 SCO는 리눅스의 신뢰성에 대하여 뻔뻔스러운 거짓말을 함으로써 사람들을 모욕했고 이는 고객들에게 신뢰를 잃는 결과를 낳았다. SCO는 결국 웹사이트에 이를 철회한다고 발표했다.

1998년 후반, SCO는 SCO 유닉스가 리눅스 호환 계층을 가졌으며 사람들이 즐겨 사용하는 리눅스 프로그램이 SCO 유닉스 상에서도 동작한다고 언론에 발표하였다. 그에 대한 반응은 매우 회의적이었다. 겨우 무료 경쟁자와 호환성을 가지려고 왜 OS에 돈을 쓰겠는가?

SCO는 오픈 소스 운동으로부터 혜택을 받을 수 있는 독특한 위치에 있다. SCO는 몇 가지 매우 가치 있는 지적 자산을 가지고 있으며, 이를 적극적으로 활용하면 오

픈 소스의 미래에 진정 강력한 위치를 차지할 수 있다. 하지만 그들은 먼저 오픈 소스에 대한 믿음을 가져야 한다. 오픈 소스를 SCO의 지적 자산을 파괴하는 위협으로 보지 말고 지적 자산에 혁신을 가져올 기회로 보아야 한다.

물론 오픈 소스에 대해 SCO 또는 심지어 썬과 같은 회사가 보여준 전술은 마이크로소프트사의 행동에 비하면 미약하다. 지금까지도 마이크로소프트는 자신의 독점 모델에 갇혀 있으며, 최소한 윈도우 2000 발표를 통해 관철하려고 결심한 듯 보인다.

우리가 예상하기에 윈도우 2000은 엄청난 광고를 하면서 2000년 후반이나 2001년 초에 발매될 것이다. 윈도우 2000의 발매는 NT와 98을 결합하는 엄청난 사건이 될 것이다. 이 사건 전후나 6개월 이전쯤에 새로운 마이크로소프트 윈도우 시스템에 대한 발표가 있을 것이다. 마이크로소프트는 항상 ‘확실히 돈 되는 엔터프라이즈 시장’, 즉 회사의 핵심 자료를 처리하는 영역을 탐내왔다. 하지만 지금까지 마이크로소프트가 윈도우 NT 또는 윈도우 2000 시스템을 이 시장의 영역에서 원하는 만큼의 안정성을 가지고 공급한 사례는 없었다. 따라서 마이크로소프트사는 이 새로운 시스템이야말로 다가오는 솔루션이라고 공포할 것이다.<sup>03</sup>

환상적인 변화를 대표할 이 제품을 ‘윈도우 엔터프라이즈’ 또는 WEnt라고 부르도록 하겠다. 마이크로소프트는 NT를 보면서 근본적으로 NT를 어떻게 더 신뢰받고 안정적이게 할 수 있는지 물을 것이다. 리누스 토르발스가 그의 글에서 지적하듯 OS 이론은 지난 20년간 크게 바뀐 것이 없으며, 따라서 마이크로소프트 엔지니어는 처음으로 다시 회귀하여 실행 레벨을 지저분하게 건드리지 않고 깨끗하고 탄탄하게 결합되도록 작성한 커널이야말로 안정성과 속도를 이룰 수 있는 가장 좋은 방법이라고 말하게 될 것이다. 윈도우 NT 커널의 주요 버그, 즉 제대로 테스트하지 않았거나 잘못 선택된 서드파티 드라이버를 포함하고 GUI 부분을 커널 일부로 만든 점을 고치려면 마이크로소프트는 어마어마하게 느린 에뮬레이션 계층을 만들거

나 수많은 애플리케이션과의 호환성을 포기해야 한다. 마이크로소프트는 분명히 이 어떤 과정도 추구할 수 있다. 소프트웨어를 구매하는 기업에서는 마이크로소프트가 리눅스처럼 안정적인 커널 아래 이미 가능한 것들을 제공할 정도로 믿음직한지 자문하게 될 것이다. 하지만 이미 오픈 소스 프로그램들은 충분히 성숙하여 있다.

물론 시간이 답을 말해줄 것이다. 마이크로소프트가 이런 수준의 튼튼하고 안정적인 소프트웨어를 실제로 만들 수 있을지는 아무도 모른다. 에릭 레이먼드는 ‘핼러윈 문서’를 예로 들어 마이크로소프트 내부에서도 이에 대한 의문을 가지고 있다고 말한다.

## 오픈 소스에 대한 위협

대부분의 과학 분야 엔터프라이즈와 마찬가지로 소프트웨어 벤처 대부분은 실패한다. 밥 영이 지적한 것처럼 성공적인 오픈 소스 소프트웨어를 만드는 것은 성공적인 독점 소프트웨어를 만드는 것과 크게 다르지 않다. 두 경우 모두 진짜 성공은 드물며, 가장 훌륭한 혁신자는 실수에서 배우는 자다.

과학과 소프트웨어 분야 모두 혁신을 가져다주는 자유분방한 창조성에는 대가가 따른다. 활발한 오픈 소스 프로젝트를 제어하는 일은 힘들다. 따라서 제어를 잊을지 모른다는 불안감 때문에 일부 개인과 많은 기업이 활발히 참여하기를 꺼린다. 특히 오픈 소스 소프트웨어 프로젝트를 시작하거나 참여할 때의 걱정거리 하나는 커다란 경쟁자나 많은 사람이 끼어들어 코드에 분열(fork, 유닉스 프로세스가 두 개로 나뉘도록 하는 함수)을 초래할지 모른다는 염려다. 마치 두 갈래 길이 나오는 것처럼 코드 기반이 종종 전혀 다르고 호환성이 없는 길로 나뉘어 영영 만나지 못 할 수도 있다. 이는 간과할 문제가 아니다. 예를 들어 BSD 기반의 운영체제가 여러 번 분열하여 NetBSD, OpenBSD, FreeBSD로 나뉜 경우를 볼 수 있다. 그러나 리눅스에서 이런 일이 일어나지 않는 이유는 무엇일까?

이런 일이 일어나지 않도록 지켜주는 것 하나는 리눅스 커널 개발에 사용되는 개방 방식이다. 리눅스 토르발스, 앨런 콕스 Alan Cox, 그리고 나머지 몇몇 사람이 탄탄하게 운영하고 있으며, 커널에 기능을 추가하고 접근하는 중앙 집권적 권위를 가지고 있다. 리눅스 커널 프로젝트는 리눅스라는 독재자가 있는 자비로운 독재라고 불려 왔다. 지금까지 이 모델은 커널 안에 쓸데없는 부차적인 기능 없이 탄탄하고 잘 짜여진 커널을 만들어왔다.

아이러니하게도 리눅스에서 실제 코드 분열은 거의 일어나지 않았지만, 중요 장치를 제어하기에 알맞도록 리눅스를 하드 리얼타임 커널로 만들어주는 패치와 같은 것이 등장하였으며, 또한 정말로 희한한 아키텍처에서 실행되는 리눅스 버전도 생겨났다. 원본 커널에 기초하고 있으며 그로부터 파생된 것이기 때문에 패치를 코드 분열이라고 볼 수도 있지만, 각자 리눅스의 특정 활용 영역만을 차지하고 있으므로 전체적으로는 리눅스 공동체에 분열 효과를 주지는 않는다.

비유를 통해 특별한 경우에 적용되는 과학적 방법을 생각해보자. 대부분 세상은 뉴턴의 운동 법칙만 가지고도 역학 계산을 충분할 수 있다. 큰 질량 또는 높은 속도라는 특별한 조건에서만 우리는 아인슈타인의 상대성 이론에 의지하면 된다. 오래된 뉴턴의 이론 기반을 훼손시키지 않고도 아인슈타인의 이론은 성장하고 확장 할 수 있다. 그러나 소프트웨어 벤치는 마치 과학적 이론이 그러하듯 경쟁 과정 중에 빈번히 충돌한다. 루시드 Lucid의 역사를 보자. 루시드는 인기 있는 프로그래밍 편집기인 Emacs<sup>Emacs</sup>를 활용하여 현대적인 버전을 만들고 리처드 스톤먼이 만든 원판 Emacs 대체품을 개발자에게 판매한 회사였다. 이 루시드사의 대안은 루시드 Emacs라고 불렸고 그다음에는 XEmacs라고 불렸다. 루시드 팀이 여러 회사에 XEmacs를 판매하려고 다가갔지만 XEmacs와 Emacs 사이의 확실한 결과 차이를 이끌어낼 수 없었다. 게다가 그 당시 컴퓨터 시장이 활기를 띠지 않은 상황도 작용하여 루시드는 오래가지 못했다.

흥미롭게도 루시드는 사업을 포기하기 전에 XEmacs 코드를 GPL화했다. 이 실패한 사업의 예도 오픈 소스 소프트웨어의 오랜 수명을 말해준다. 사람들이 소프트웨어가 유용하다고 생각하는 한, 새로운 시스템과 아키텍처에 맞도록 유지할 것이다. 심지어 XEmacs는 지금도 엄청난 인기를 누리고 있으며, Emacs 해커들에게 어떤 Emacs를 더 좋아하느냐고 물으면 재미있는 논쟁이 불붙는 것을 볼 수 있다. 즉, 현재의 XEmacs는 매우 적은 사람만이 참여하는 오픈 소스 프로젝트지만 여전히 새로운 시대와 아키텍처에 맞게 변화하고 적응하는 진보적인 제품의 예다.

## 오픈 소스 해커에게 동기 부여하기

루시드의 경험은 프로그래머들이 종종 직접적인 보상 이상의 가치를 갖는 프로젝트에 헌신적인 애정을 갖는다는 사실을 보여준다. 왜 사람들은 자유 소프트웨어를 만드는가? 시간당 몇 백 달러씩 받을 수 있는 소프트웨어를 왜 그냥 무료로 내주는가? 이로부터 그들이 얻는 것은 무엇인가?

단지 이타주의만이 동기는 아니다. 오픈 소스에 이바지하는 사람들은 마이크로소프트의 스톡옵션을 받지는 않지만, 집세를 내거나 아이들을 양육할 수 있는 돈을 벌 기회를 보장해 주는 명성을 쌓았다. 밖에서 보면 모순으로 보일 수도 있겠지만 어쨌든 자유 소프트웨어만으로 먹고 살 수는 없다. 그 답은 일과 보상이라는 상투적인 것을 넘어서는 선상에 있다. 우리는 그냥 새로운 문화만이 아닌 새로운 경제 모델이 제 모습을 갖춰 가는 것을 목격하는 것이다.

에릭 레이먼드는 오픈 소스 공동체에 있어 일종의 자칭 참여 인류학자가 된 사람이다. 그의 글에서는 왜 사람들이 소프트웨어를 개발하여 그냥 공개하는지에 대한 이유를 다룬다. 대부분의 경우 이 사람들은 몇 년간 코딩을 해왔던 사람들이며 프로그래밍 자체를 짐이나 일로 생각하지 않는다는 점을 명심하자. 아파치 또는 리눅스 커널과 같은 매우 복잡한 프로젝트는 지적인 훈련에 있어 최고의 만족감을 가져준다. 달리기 선수가 경주를 위해 달리는 동안 느끼는 흥분처럼 진정한 프로그래머

는 완벽한 루틴이나 깔끔한 코드를 작성한 후 그와 비슷한 흥분을 느낀다. 며칠 동안 골칫덩이였던 복잡한 코드를 완성하고 디버깅한 후 느끼는 환희는 말로 표현할 수가 없다.

중요한 점은 많은 프로그래머가 단지 좋아하기 때문에 코드를 짜며, 사실 이것이 바로 자기의 지적 능력을 정의하는 방식이라는 것이다. 코딩하지 않는다면 프로그래머는 경쟁할 기회를 박탈당한 운동선수처럼 무기력함을 느낄 것이다. 운동선수와 마찬가지로 프로그래머에게 문제는 훈련이다. 사실 많은 프로그래머는 일단 코드를 마스터한 후에는 코드 유지에 흥미를 느끼지 않는다.

또한 어떤 프로그래머는 자신의 기술에 대하여 이러한 ‘당당한’ 관점을 갖지 않고 학자적인 관점을 갖고 있다. 많은 프로그래머는 대체로 자신을 과학자라고 생각한다. 과학자들이란 자신의 발명으로부터 이익을 취하지 않으며, 모든 사람이 발명으로부터 혜택을 입을 수 있도록 공표하고 공유한다. 과학자란 지식의 추구를 통해 이익을 추구하는 사람이 아니다.

프로그래밍에 대한 이러한 모든 관찰에서 공통점이 있다면 그것은 명성에 대한 강조다. 프로그래밍은 주고받는 선물 문화다. 프로그래머가 이룩한 작업의 가치는 오로지 다른 사람과 공유하는 데서 나온다. 일이 널리 공유될수록 그 가치는 더욱 커지며 그냥 미리 컴파일된 바이너리에 의한 결과만이 아니라 소스를 보여줌으로써 좀 더 완전히 공유하게 될 때 그 가치는 더욱 커진다.

프로그래밍이란 또한 에릭 레이먼드가 “가려운 곳을 긁는다”고 표현하듯 자아성취에 관한 것이다. 오픈 소스 프로젝트 대부분은 불만족에서 나온다. 작업에 필요한 도구를 찾는데 하나도 찾지 못했거나, 찾았더라도 버그가 있거나 유지가 제대로 되지 않을 때가 있다. 에릭 레이먼드의 fetchmail, 래리 월 Larry Wall의 펄 Perl, 그리고 리눅스 토르발스의 리눅스가 이런 식으로 시작되었다. 여러모로 자아성취는 스톤헌이 GNU 프로젝트를 시작하게 한 동기의 기초를 이루는 중요한 개념이다.

## 리눅스 벤처와 미래 투자

해커의 동기는 지적이겠지만 꼭 그 결과가 회생을 요구할 필요는 없다. 점점 더 많은 기업과 개별 오픈 소스 프로그래머들이 실용주의와 기회를 염두에 두고 등장하고 있다.

우리가 일하고 살아가는 실리콘밸리에는 이 지역 경제에 동력을 제공하는 투자와 벤처의 역사가 있다. 이 역사는 상업적인 이득을 위해 트랜지스터를 맨 처음 활용하고 마이크로프로세서가 산업의 동력이 되어 성가신 회로 기판을 수천 개의 칩과 개별 트랜지스터로 대체하던 시절까지 거슬러 올라간다.

언제 어느 때든 벤처캐피털이 집중하는 새로운 기술이 있다. 다른 회사나 기회를 무시하는 것은 아니지만, 벤처캐피털은 경제적인 성과를 위해서 단지 성공적인 기업이 아니라 3년 안에 초기 주식 공개IPO를 할 수 있거나 오라클이나 시스코 등의 회사에 몇 천만 달러에 팔 수 있는 주목받는 그런 최신 기업을 필요로 한다.

1998년 인터넷의 거대한 물결이 퇴조하고 있다. 즉, 넷스케이프사의 화려한 데뷔로부터 시작된 인터넷 주식공개의 열기가 수그러들기 시작했다는 뜻이다. 딱 어울리는 상징적인 행동으로 아메리카 온라인이 넷스케이프사를 인수했다는 사실이 이 세대의 종말을 잘 보여준다. 인터넷 주식은 현재 투자자들이 더욱 면밀하게 검토하고 있으며 일반적으로 인터넷 회사들은 그 밖의 다른 회사와 같은 기준을 가진다. 투자자들은 투자에 앞서 그럴듯한 수익성을 예상할 수 있어야 한다.

그럼 벤처는 어디로 갈 것인가? 우리의 예측으로는 리눅스와 오픈 소스 소프트웨어와 관련된 회사가 20세기 말을 전후하여 가장 주목받는 투자 대상이 될 것이다. 여러분은 1999년 후반부터 레드햇 소프트웨어를 시작으로 리눅스와 오픈 소스 분야의 활발한 주식 공개 현상을 목격하게 될 것이다. 투자액은 사람들이 놀랄 만큼의 액수일 것이며, 스크립틱스 Scriptics, 센드메일 Sendmail, Vix.com은 회사가 가진

시장 조건에 주목하고 몰려드는 투자자들의 꿈의 회사가 될 것이다. 사실 의문은 벤처캐피털이 오픈 소스 쪽으로 들어올 것인가의 문제보다 왜 벌써 그러한 흐름이 시작되었는가에 있다. 자유 소프트웨어란 새로운 것이 아님을 명심하자. 리처드 스톤먼은 1984년 FSF를 시작하였고 훨씬 이전까지 거슬러 올라가는 전통을 만들었다. 지금처럼 인기를 끄는 데 왜 그렇게 오랜 시간이 걸렸는가?

컴퓨팅 환경을 살펴보면 자금이 풍부하고 거대한 회사들이 시장의 막대한 지분을 장악한다는 사실을 알 수 있다. 실리콘밸리에서 에인절 혹은 벤처 투자자의 후원을 찾는 유망한 애플리케이션 업체는 마이크로소프트사에 반대되는 위치에 놓이면 자금 지원을 받을 수 없다는 사실을 금방 깨달을 수 있다. 따라서 모든 시작은 결국 마이크로소프트의 게임을 하느냐 아니면 전혀 안 하느냐다.

숨 막히는 이 환경이 바로 자유 소프트웨어가 떠오르게 한 원동력이라는 것은 분명하다. 마이크로소프트 윈도우 운영체제에서 작동하는 프로그램을 만드는 프로그래머라면, 누구나 마이크로소프트 윈도우 위에서 실행되는 모든 프로그램이 마이크로소프트 라이브러리에 완벽하게 의존하도록 만든 잡다한 인터페이스 집합이라고 말할 것이다. 즉, 마이크로소프트가 프로그래머들에게 제시하는 인터페이스는 윈도우 전용 프로그램들을 다른 운영체제에 이식하기 어렵게 만드는 목적에 부합한다는 뜻이기도 하다.

마이크로소프트가 아직 점령하지 못한 가장 큰 경기장인 인터넷은 그러한 제약을 갖고 있지 않다. 스캇 브래드너 Scott Bradner가 밀하는 바로는, 인터넷은 회사의 자본력이 아니라 참여하는 개개인 모두를 위해 유지되는 개방 표준의 강력한 집합 위에 건설되어 있다. 여러모로 인터넷은 원래 오픈 소스 벤쳐다. 인터넷은 튼튼한 개방 표준에 기초하도록 함으로써 폭넓고 다양한 프로그래머들이 인터넷 애플리케이션을 개발할 수 있게 되었다. 인터넷의 놀라운 성장은 개방 표준 모델의 힘을 극명하게 보여주는 예다.

인터넷의 성공에 내재하는 고유 구조가 오픈 소스 운동에도 존재한다. 레드햇, SuSE와 같은 리눅스 배포 사업자가 서로 경쟁하는 것이 그렇다. 하지만 그들은 개방 표준과 공유한 코드 위에서 경쟁한다. 예를 들어 리눅스 배포 사업자는 서로 다른 패키지 관리 시스템을 사용하여 개발자들을 가둬두는 대신 둘 다 레드햇 패키지 관리자 RPM를 패키지 관리 도구로 사용한다. 데비안 Debian은 다른 패키지 관리 도구를 사용하지만 데비안과 레드햇의 패키지 관리 도구는 둘 다 오픈 소스 프로그램이므로 둘 사이의 호환성이 유지됐다. 즉, 인터넷 기술을 벤처 투자자에게 매력적인 격전장으로 만들어 준 내부 구조가 오픈 소스에도 있으며, 따라서 오픈 소스 기술도 똑같이 매력적으로 만들어 줄 것이라 기대할 수 있다.

그러나 더욱 중요한 점은 오픈 소스가 적극적으로 활용할 수 있는 새로운 내부 구조를 인터넷이 만들어 왔다는 것이다. 현시점은 소프트웨어 엔터프라이즈 시대에서 팀 오라일리 Tim O'Reilly가 묘사하는 인포웨어 inforware 엔터프라이즈의 시대로 이동하는 단계다. 이러한 이동을 하기 위해서는 진입 장벽과 배포 비용을 극적으로 낮춰야 하며, 바로 인터넷이 이러한 장벽을 낮추는 데 이바지하였다.

## 과학과 새로운 르네상스

오늘날의 오픈 소스 개발 모델은 사반세기 전의 학문적인 컴퓨터 과학에 그 뿌리를 둔다. 그러나 오늘날 오픈 소스를 정말로 성공적으로 만들어준 요인은 정보의 신속한 확산을 가능케 해준 인터넷이다. 웃슨과 크리이 이종 구조를 발견했을 때, 그들은 정보가 캠브리지에서 칼 테크까지 전송되는 데 며칠 또는 몇 주 정도 걸릴 것이라고 예상했다. 오늘날에는 정보의 전송이 거의 동시다. 현재 과학의 발전이 르네상스 시대에 발명된 출판 기술에 의해 가능했던 것처럼 오픈 소스는 인터넷이 가능케 한 디지털 르네상스에서 태어났다.

중세 시대에는 마땅한 정보 내부 구조가 없었다. 저작물은 큰 노력을 들여 손으로 복사해야 했으므로 정보 그 자체에 가치가 있어야 했다. 교역 자료, 은행 업무, 외

교 문서와 같은 것을 예로 들 수 있다. 이 정보는 간략했고 전송할 만한 가치를 가지고 있다. 연금술사, 신부, 철학자 등 나중에 과학자로 불린 사람들의 이론적인 작업은 우선권이 낮았으며 더더욱 느리게 전파되었다. 출판업은 정보 내부 구조에 접근할 수 있는 장벽을 획기적으로 낮춤으로써 이 모든 것을 바꾸어 놓았다. 서로 고립되어 연구해왔던 학자들은 처음으로 전 유럽에 걸쳐 존재하는 다른 학자들과 일종의 공동체를 형성할 수 있었다. 하지만 이러한 공동체를 건설해나가는 작업은 정보를 완전히 공유하겠다는 절대적인 서약을 필요로 했다.

이 공동체로부터 학문적인 자유, 그리고 현재 과학적 방법이라고 부르는 과정이 탄생하였다. 공동체를 형성해야 할 필요성이 없었다면 이러한 일은 불가능했을 것이다. 그리고 수세기에 걸친 정보의 개방된 공유는 과학적 공동체를 서로 결합해주는 시멘트 역할을 했다.

만약 뉴턴[Newton]이 운동 법칙을 발표하지 않고 30년 전쟁의 포병 청부업자로 사업을 시작했다고 생각해보라. “싫소, 내가 어떻게 포물선 궤도를 알았는지 당신에게 알려주지 않을 것이오. 대신 당신의 포를 조정해 주는 데 비용을 받겠소.”라고 고집했다면 어떻게 되었을까? 물론 이런 생각은 터무니없어 보인다. 과학은 이런 식으로 발전하지 않았으며 발전할 수도 없기 때문이다. 만약 과학을 생각했던 사람들이 이런 생각을 가졌었다면 비밀 유지 때문에 과학이 만들어지거나 진화할 수 없었을 것이다.

인터넷은 디지털 시대의 출판업이다. 이를 통해 다시 한 번 정보 내부 구조의 장벽이 획기적으로 낮아졌다. 소스 코드를 유닉스 오리지널 버전처럼 종이테이프로 배포하거나 DOS 시절의 플로피, 심지어 CD-ROM 형태로 배포할 필요가 없다. 대신에 FTP 또는 웹 서버가 저렴하면서도 동시성을 갖는 배포 지점이 될 수 있다.

이 르네상스가 커다란 약속을 제시하지만 오픈 소스 모델이 기반을 두는, 수세기를 이어온 과학적 유산을 잊어서는 안 된다. 오늘날 컴퓨터 과학과 컴퓨터 산업은 불

편한 동맹 관계 속에 존재한다. 여기에는 마이크로소프트와 같은 거대한 기업이 단기적인 이익을 위해 새로운 제품을 독점화하려는 압력이 존재한다. 하지만 점점 더 많은 컴퓨터 과학 분야의 개발 작업이 학문 분야가 아닌 산업 분야에 뿌리를 두게 됨에 따라, 컴퓨터 산업은 아이디어의 공개적인 공유, 즉 오픈 소스 개발 모델을 통해 컴퓨터 과학에 자양분을 공급해야 한다. 컴퓨터 산업은 대의명분을 위한다는 이타적인 동기가 아니라 가장 기본적인 실용적 이유, 계몽된 이기심을 위해 이렇게 해야 한다.

우선 금전적인 보상이 오픈 소스의 최고 프로그래머들이 갖는 주요 관심사라고 믿는 것은 컴퓨터 산업 분야의 근시안적인 생각이다. 이들을 산업과 연계시키려면 그들이 우선으로 생각하는 것을 존중해야 한다. 이들은 명예<sup>reputation</sup> 게임에 관여 한다. 그리고 역사는 과학적인 성공이 물질적인 성공보다 지속된다는 것을 보여주었다. 지난 백 년에 걸쳐 살아온 기업가 중 우리가 기억하는 사람이라곤 카네기 Carnegie, 록펠러 Rockefeller 등 극소수다. 반면에 우리는 지난 백 년에 걸쳐 살아온 수 많은 과학자와 발명가(아인슈타인, 에디슨, … 폴링)를 기억한다. 이 시대의 역사가 지금으로부터 백년 뒤 쓰여진다면 사람들은 아마도 빌 게이츠라는 이름과 그 밖의 몇 명만 기억할 것이지만, 리처드 스톤먼이나 리눅스 토르발스와 같은 이름은 아마도 더욱 잘 기억하게 될 것이다.

두 번째로 더욱 중요한 것은 산업이란 과학에서 제공하는 혁신을 필요로 한다는 것이다. 오픈 소스는 과학의 속도와 창조성을 가지고 새로운 소프트웨어를 개발하고 고쳐나갈 수 있다. 컴퓨터 산업은 오픈 소스 개발에서 생겨날 다음 세대의 아이디어를 필요로 한다.

리눅스의 예를 다시 한 번 생각해보자. 리눅스는 마이크로소프트가 윈도우 NT를 개발하기 시작한 후로 약 5년쯤 후에 시작된 프로젝트다. 마이크로소프트는 수많은 시간 및 인력과 수백만 달러를 윈도우 NT 개발에 쏟아부었다. 그러나 오늘날 리

눅스는 PC 기반의 서버 시스템에서 윈도우 NT의 경쟁적 대안으로 받아들여지고 있으며 오라클, IBM, 그리고 기타 주요 엔터프라이즈 소프트웨어 업체가 주요 미들웨어와 백엔드 소프트웨어와의 이식성에 힘을 쓴고 있다.

오픈 소스 개발 모델이 아니었다면 마이크로소프트와 같은 권력과 자원을 필요로 하는 소프트웨어만이 세상에 가득했을 것이다. 즉, 디지털 르네상스를 지속하기 위해서는 오픈 소스 개발이 필요하며, 오픈 소스 개발은 컴퓨터 과학뿐만 아니라 컴퓨터 산업에서도 진보를 이끌어 나가고 있다.

---

- 01 역자주\_원어는 the Force로 영화 '스타워즈'에서 동양적인 사고인 기(氣)와 비슷한 개념으로 사용된다. 이 글의 여러 부분에서는 미국에서 잘 알려진 '스타워즈'의 대사를 사용한다. 따라서 원어와 영화상의 복합적인 의미를 살리기 위해 '포스'라고 번역했다.
- 02 역자주\_루크는 영화 '스타워즈 에피소드 4, 새로운 희망' 편부터 나오는 다크베이더(Darth Vader)의 아들이다. 다크베이더도 가르쳤던 제다이(Jedi) 기사 오비완(Obiwan)이 루크에게 포스(the Force)를 사용이라고 훈계하는 대목을 조금 바꾼 것이다.
- 03 편집자주\_윈도우 2000은 2000년 2월에 출시되었고, 현재는 호평 반, 비판 반인 상태다.

# 1 | 해커 문화의 짧은 역사

에릭 레이먼드(Eric S. Raymond)

최준호 역

## 프롤로그 -진정한 프로그래머

태초에 진정한 프로그래머들이 있었다.

그들은 자신을 스스로 그렇게 부르지 않았다. 또한 ‘해커’라고 부르지도 않았고 특별히 어떤 명칭이 있는 것도 아니었다. ‘진정한 프로그래머’라는 별명은 1980년 이후까지 만들어지지도 않았다. 그러나 1945년 이후부터 컴퓨팅 테크놀러지 는 전 세계의 총명하고 창조적인 수많은 사람을 유혹하였다. 엑터트 Eckert와 모출리 Mauchly의 애니악 ENIAC에서부터 소수 열정적인 프로그래머에 의해 자의식 강한 기술 문화가 계속되었으며, 사람들은 소프트웨어를 재미로 만들고 즐기기 시작했다.

진정한 프로그래머는 전형적으로 공학이나 물리학을 배운 사람들이었다. 이들은 흰 양말과 폴리에스터 셔츠와 넥타이 차림에 두꺼운 안경을 낀 채 기계어, 어셈블리어, 포트란, 그리고 이제는 잊혀진 몇몇 고대 언어로 프로그래밍하였다. 이들이 해커 문화의 선각자들이며 대부분 본격적인 컴퓨터의 역사가 시작되기 이전이었기 때문에 찬양을 받지 못한 주인공들이었다.

2차 세계대전이 끝난 후, 1970년대 초반까지 배치 batch 컴퓨팅과 ‘거대한 고철 덩어리’인 메인프레임이 주름잡던 때에 진정한 프로그래머들은 컴퓨팅에 있어 기술 문화를 주도하였다. 잘 알려진 Mel 이야기(Jargon File에 포함되어 있다)와 여

러 가지 머피의 법칙, 여전히 많은 컴퓨터 룸을 기품있게 해주는 가짜 독일식의 ‘Blinkenlights’ 포스터까지 경배하던 해커 전승(傳承)의 일부는 이 시대까지 거슬러 올라간다.

‘진정한 프로그래머’ 문화에서 자란 사람들은 1990년대까지도 활동적이었다. 크레이Cray 슈퍼컴퓨터 제품군의 설계자인 세이모어 크레이Seymour Cray는 직접 설계한 전체 운영체제를 자신이 설계한 컴퓨터에 탑재했다고 말했다. 이는 8진수를 사용했으며 오류 없이 동작하였다. 진정한 프로그래머의 위대함은 이런 것 아닐까?

더 가벼운 내용으로 ‘악마의 DP 사전(‘The Devil’s DP Dictionary’, McGraw-Hill, 1981)’의 저자인 스탠 켈리 부틀Stan Kelly-Bootle과 뛰어난 전승자들은 1948년 최초의 동작하는 저장 프로그램 디지털 컴퓨터인 맨체스터 마크 I[Manchester Mark I]에서 프로그래밍하였다. 최근 그는 컴퓨팅 잡지에 종종 오늘날 해커 문화에 대한 활발하고 박식한 대화의 형태를 띠는 기술적인 유머 칼럼을 기고하고 있다.

다른 사람들, 가령 데이비드 런스톰David E. Lundstrom은 이러한 초기 시대의 일화에 대한 이야기를 썼다(‘A Few Good Men From UNIVAC’, 1987).

‘진정한 프로그래머’ 문화가 해낸 것은 대화형 컴퓨팅, 대학, 네트워크의 발달이었다. 그들은 결국 오늘날의 오픈 소스 해커 문화로 진화할 계속되는 공학의 전통을 탄생시켰다.

## 초기 해커들

오늘날 우리가 알고 있는 해커 문화의 시초는 MIT가 최초의 PDP-1을 구입한 1961년으로 생각할 수 있다. MIT 테크 모델 철도 클럽Tech Model Railroad Club, TMRC은 이 기계를 자신들이 좋아하는 기술적인 장난감으로 택하고 오늘날 우리가 인식하는 프로그래밍 도구, 관련 은어, 전체 주위 환경을 만들어냈다. 이러한 시대는 스티븐 레비Steven Levy가 쓴 ‘Hackers(Anchor/Doubleday, 1984. 번역본 ‘해커 그 광기와 비밀의 기록’, 사민서각, 1996)’에서 설명하고 있다.

MIT의 컴퓨터 문화는 ‘해커’라는 용어를 최초로 도입한 것으로 보인다. TMRC의 해커는 1980년 초기까지 전 세계의 인공지능 연구를 이끄는 센터였던 MIT 인공지능 연구소의 핵심이 되었다. 그리고 그 영향력은 ARPAnet 최초의 해인 1969년 이후 더 넓게 퍼졌다.

ARPAnet은 최초의 대륙 간 고속 컴퓨터 네트워크였다. 국방성의 디지털 통신 실험으로 만들어졌지만 수백 군데의 대학과 방위 산업체와 연구소를 연결하게 되었다. 연구자들은 어떤 곳에서든지 전례 없던 속도와 유연성으로 정보를 교환할 수 있었으며, 이는 협동 작업에 큰 후원이 되었고 기술적 진보의 속도와 강도를 크게 증가시켰다. 그러나 ARPAnet은 다른 역할도 수행했다. (그 전자 고속도로는) 미국 내의 모든 해커를 결집시켰던 것이다. 해커들은 자신들의 비영속적인 지역 문화를 개발하는 독립적 소규모 그룹으로 남는 대신, 자신들이 네트워크 부족임을 발견(또는 재발견)하였다.

최초의 의도적인 해커 문화의 유물(최초의 은어 목록, 최초의 풍자 작품, 해커 윤리에 대한 자의식적 논의)은 모두 초기 시대에 ARPAnet에서 떠돌아다니던 것이었다 (중요한 예로, Jargon File의 최초 버전은 1973년으로 거슬러 올라간다). 해커 문화는 네트워크에 연결된 대학에서 자라났고, 특히 (모든 대학에서는 아니지만) 대학의 전산과학학부에서 주로 자라났다.

문화적으로 MIT의 인공지능 연구소는 1960년 후반 다른 대학 중 최초였다. 그러나 스탠퍼드 대학의 인공지능 연구소 Stanford Artificial Intelligence Laboratory, SAIL과 카네기 멜론 대학 CMU도 이와 비슷하게 중요한 비중을 차지했다. 이 모든 곳은 전산 과학과 인공지능 연구가 번창하는 중심지였다. 이들은 기술적이고 전승적인 수준 모두에서 해커 문화에 중요한 것들에 이바지할 수 있는 총명한 사람들을 끌어들였다.

그러나 그다음이 무엇인지 이해하려면 컴퓨터 자체를 달리 살펴봐야 할 필요가 있는데, 연구소의 흥망은 컴퓨팅 기술의 변화하는 물결에 의해 주도되었기 때문이다.

예를 들어 PDP-1의 시대부터 해커 문화의 운명은 DEC(Digital Equipment Corporation)의 PDP 마이크로컴퓨터 시리즈와 함께 엮여 있었다. DEC는 상용 대화형 컴퓨팅과 시분할 운영체제를 개척했다. 이 회사의 기계들은 유연하고 강력했고, 당시에는 비교적 저렴하였으므로 많은 대학이 이 기계를 구입했다.

저렴한 시분할 시스템은 해커 문화가 자라나는 매개체였으며, ARPAnet의 시스템 대부분은 주로 DEC 기계의 네트워크 형태였다. 이들 중 가장 주목받은 것은 1967년에 처음 발표한 PDP-10이었다. PDP-10은 거의 15년 동안 해커 문화에서 선호한 제품이었다. 또한 TOPS-10(타 기계용 DEC의 운영체제)과 MACRO-10(어셈블러)은 많은 은어와 전승 속에서 여전히 향수를 느낄 수 있는 제품들로 기억되고 있다.

MIT는 다른 사람들처럼 같은 PDP-10을 사용했지만 조금 다른 길을 택하였다. 그들은 PDP-10용 DEC 소프트웨어를 완전히 버리고 지금은 전설로 남은 자신만의 운영체제 ITS를 만들어냈다.

ITS는 ‘비호환적 시분할 시스템’(Incompatible Timesharing System)을 나타내며 이는 자신만의 방식을 원했던 MIT의 태도를 잘 알 수 있게 한다. 다행히도 MIT 사람들은 자신의 교만에 상당하는 지성을 갖고 있었다. 항상 그랬지만 변덕스럽고 별나며 종종 베그도 있었던 ITS는 멋진 일련의 기술적 진보를 뒷받침하였으며, (논쟁의 여지는 있지만) 여전히 가장 오랫동안 사용된 시분할 시스템이라는 기록을 보유하고 있다.

ITS 자체는 어셈블러로 작성했지만 많은 ITS 프로젝트는 AI 언어인 LISP로 작성되었다. LISP는 당시 어떤 언어보다도 강력하고 유연하게 설계되어 25년이 지난 오늘날에도 여전히 대부분의 언어보다 뛰어날 정도다. LISP는 ITS의 해커들이 특이하고 창조적인 방식으로 자유롭게 생각하도록 하였다. 바로 이것이 성공의 주요 요인이 되었으며, LISP는 여전히 해커 문화가 좋아하는 언어 중 하나로 남아 있다.

ITS 문화의 많은 기술적 창조는 오늘날에도 살아있다. Emacs 프로그램 편집기가 아마도 제일 잘 알려진 것이며, Jargon File에서 볼 수 있듯이 ITS의 전승 중 많은 것들이 여전히 해커들에게 ‘살아’ 있다.<sup>01</sup>

SAIL과 CMU도 잠자코 있지만은 않았다. SAIL의 PDP-10으로 성장한 큰 해커들의 중심인물 중 많은 사람이 나중에 개인용 컴퓨터와 오늘날의 윈도우·아이콘·마우스 소프트웨어 인터페이스 개발의 핵심 인물이 되었다. 또한 CMU의 해커들은 전문가 시스템과 산업적 로봇공학에서 최초의 실용적인 대규모 애플리케이션을 이끌게 될 작업을 하고 있었다.

이 문화와 관련된 또 다른 중요한 장소는 Xerox PARC의 유명한 팔로 alto 연구소 Palo Alto Research Center다. 1970년 초반부터 1980년 중반까지 10년 이상 PARC는 하드웨어와 소프트웨어의 놀라운 혁신을 이루었다. 현대의 마우스, 윈도우, 아이콘 스타일의 소프트웨어 인터페이스가 여기서 발명되었다. 레이저 프린터, 근거리 네트워크 LAN도 그랬다. PARC의 D 머신 시리즈는 10년 전에 1980년대의 강력한 개인용 컴퓨터를 예상한 것이었지만, 슬프게도 이 예언자들은 자기 회사에서는 영예를 얻지 못했다. 그랬기 때문에 PARC를 자신을 제외한 모든 사람을 위한 뛰어난 아이디어를 개발하는 곳으로 특징짓는 것이 일반적인 농담이 되었다. 해커 문화에 대한 그들의 영향은 대단한 것이었다.

ARPAnet과 PDP-10 문화는 1970년대에 걸쳐 크고 다양하게 자라났다. 대륙 간 동호인들 사이의 협동을 돋는 데 사용된 전자 메일링 리스트 기능은 점점 더 많은 부분이 사회적이거나 오락적인 목적으로 사용되었다. DARPA는 일부러 기술적으로 ‘인증받지 않은’ 모든 활동을 모른 체했으며, 그들이 부담해야 할 별도의 과부하는 뛰어난 짚은 사람을 컴퓨팅 분야에 끌어들이기 위해 들여야 할 약간의 비용이라는 점을 이해하고 있었다.

---

01 : <http://www.tuxedo.org> 참고

아마도 ‘사회적’인 ARPAnet 메일링 리스트 중 가장 잘 알려진 것은 공상 과학 팬들을 위한 SF-LOVERS일 것이다. 이 리스트는 사실 ARPAnet이 진화하여 이루어진 거대한 ‘인터넷’에서 오늘날에도 여전히 존재하고 있다. 그러나 나중에 CompuServe, Genie, Prodigy와 같은 이윤을 목적으로 운영되는 시분할 서비스 때문에 상업화된 의사소통 스타일을 개척한 다른 것들도 많이 있었다.

## 유닉스의 부상

그동안 뉴저지의 황야에서는 결국 PDP-10의 전통을 뒤엎을 만한 어떤 것이 1969년부터 진행되고 있었다. ARPAnet이 탄생한 해는 켄 톰슨Ken Thompson이라는 벨 연구소의 해커가 유닉스를 발명한 해이기도 했다.

톰슨은 ITS와 근원이 같은 Multics라는 시분할 운영체제의 개발 작업에 관여하고 있었다. Multics는 사용자와 대부분 프로그래머에게 보이지 않도록 운영체제의 복잡성이 내부에 얼마나 감추어질 수 있는지를 시험하는 무대였다. 이 아이디어는 Multics를 외부에서 (그리고 프로그래밍에서!) 더 쉽게 사용할 수 있도록 하여 더 많은 실제적인 작업이 이루어질 수 있도록 하는 것이었다. 그러나 벨 연구소는 Multics가 쓸모없는 흰 코끼리(시스템은 나중에 Honeywell에 의해 상업적으로 판매되었지만 성공하지 못하였다)처럼 비대해져 가는 징후를 보이자 프로젝트에서 손을 떼었다. 하지만 켄 톰슨은 Multics 환경을 그리워했고 쓰레기가 되어버린 DEC PDP-7에 자기 생각을 구현해보기 시작하였다.

데니스 리치Dennis Ritchie라는 해커는 톰슨의 유아기적 유닉스에서 사용하기 위해 ‘C’라고 불리는 새로운 언어를 발명하였다. 유닉스처럼 C도 즐겁고, 제한이 없고, 유연하게 설계되었다. 이 도구에 대한 관심은 벨 연구소 내에 폭쳤고, 톰슨과 리치는 그곳 내부에서 사용하기 위해 우리가 지금 사무 자동화 시스템이라고 부르는 것을 만들었던 당시인 1971년에 강력한 추진력을 얻을 수 있었다. 그러나 그들은 더 큰 목적에 관심이 있었다.

전통적으로 운영체제는 해당 호스트에서 최대한의 효율을 얻어내기 위해 어셈블리 어로 작성했다. 톰슨과 리치는 하드웨어와 컴파일러 테크놀러지가 전체 운영체제를 C로 작성할 수 있을 만큼 좋아졌다는 사실을 인식한 첫 번째 사람 중 하나였다.

이런 일은 일찍이 없었으며 이 사실이 암시하는 것은 대단한 사항이었다. 만약 유닉스가 서로 다른 형태의 기계에서 같은 모습, 같은 기능을 제공할 수 있다면 모두를 위한 공통 소프트웨어 환경으로 동작할 수 있을 것이고, 사용자들은 더 이상 기계가 쓸모없어질 때마다 소프트웨어를 새로 설계하기 위해 비용을 지불할 필요가 없을 것이기 때문이다. 즉, 해커들은 기계마다 같은 소프트웨어를 매번 새로 개발하지 않고 서로 다른 기계에서 소프트웨어 툴킷을 옮기기만 하면 되는 것이다.

유닉스와 C는 호환성 이외에 또 다른 중요한 강점이 있었다. 둘 다 “간단하고 명청하게 하자”는 철학에서 만들어졌다는 것이다. 프로그래머는 항상 매뉴얼을 참조할 필요 없이 C의 전체적인 논리 구조를 머릿속에 유지할 수 있었다(이전이나 이후의 다른 대부분 언어와는 다르게). 그리고 유닉스는 서로 유용한 방식으로 결합할 수 있도록 설계된 간단한 프로그램의 유연한 툴킷으로 구성되었다.

이 조합은 설계자가 전혀 기대하지 않았던 것들을 포함하여 아주 넓은 범위의 컴퓨팅 작업에 적합한 것으로 증명되었다. 유닉스는 공식적인 지원 프로그램이 없었지만 AT&T 내에서 아주 빠르게 퍼져 나갔다. 1980년대에 유닉스는 수많은 대학과 연구 컴퓨팅 사이트로 퍼져 나갔고 수천 명의 해커가 이곳을 고향으로 생각하였다.

초기 유닉스 문화를 일군 기계는 PDP-11과 그 후속작인 VAX였다. 그러나 유닉스의 이식성 때문에 ARPAnet 전체에서 찾을 수 있는 다양한 기계에서 거의 수정 없이 실행할 수 있었다. 그리고 아무도 어셈블리를 사용하지 않았다. 반면에 C 프로그램은 이런 모든 기계 사이에서 즉시 사용할 수 있었다.

유닉스는 유닉스 간 복사 프로토콜 Unix-to-Unc Copy Protocol, UUCP이라는 일종의 자신만의 네트워킹도 할 수 있었다. 속도가 느리고 신뢰성은 없지만, 저렴했다. 또한 어떤 유닉스 기계든 두 대 사이는 일반적인 전화선을 통해 점대점 이메일을 교환할 수 있었다. 이 기능은 외부의 별도 기능이 아니라 시스템에 내장되어 있었다. 유닉스 사이트는 자신만의 네트워크 국가를 형성하기 시작하였으며 해커 문화는 이를 따랐다. 이를 증명하듯 1980년 최초의 유즈넷은 ARPAnet보다 빠르게 성장하고 있었다.

소수의 유닉스 사이트가 ARPAnet 그 자체에 있었다. PDP-10과 유닉스 문화는 서로 만나서 주변에서 섞이기 시작하였지만 처음에는 잘되지 않았다. PDP-10 해커들은 유닉스 사람들을 LISP와 ITS의 화려하고 아름다운 복잡성에 반대되는 터무니없이 원초적으로 보이는 도구를 사용하는 건방진 한 패거리로 여기는 경향이 있었다. 그들은 불평했다. “돌칼을 들고 곰 가죽을 쓴 원시인들!”

그리고 또 다른 세 번째의 조류가 흐르고 있었다. 최초의 개인용 컴퓨터는 1975년 시장에 등장하였다. 애플사는 1977년에 만들어졌고 그 이후 거의 믿을 수 없는 속도로 성장하였다. 마이크로컴퓨터의 잠재력은 명백하였으며 또 다른 총명하고 젊은 해커 세대를 끌어들였다. 그들의 언어는 BASIC이었으며, 이는 아주 원시적이어서 PDP-10 사용자와 유닉스 애호가들 모두가 경멸할 가치조차 없는 것으로 생각하였다.

## 오랜 시대의 끝

1980년의 상황은 그랬다. 다음 세 가지의 문화는 그 가장자리에서는 겹치지만 아주 다른 기술로 구성되었다. LISP, MACRO, TOPS-10, 그리고 ITS와 결합한 ARPAnet/PDP-10 문화, PDP-11과 VAX, 전화 연결과 함께하는 유닉스와 C 그룹, 그리고 컴퓨터의 힘을 사람들에게 가져다주는 데 열중한 초기 마이크로컴퓨터 광의 무질서한 무리.

이들 중 ITS 문화는 여전히 높은 지위를 구가하고 있었지만, 곧 먹구름이 연구소를 뒤덮기 시작하였다. ITS가 의존하는 PDP-10 기술은 노쇠하였고, 연구소 자체가 인공지능 기술을 상업화하려는 첫 시도에 의해 분열이 생기기 시작했다. 최고의 연구소 직원 중 일부(그리고 SAIL과 CMU의)는 새롭게 등장한 회사의 고소득직에 유혹되어 나갔다.

그러던 중 1983년에 치명타가 될 만한 사건이 일어났다. DEC가 PDP-11과 VAX 라인에 집중하기 위해 PDP-10의 후속작을 취소하였던 것이다. ITS는 더 이상 미래가 없었다. 이식성이 없다는 이유가 가장 크게 작용하였고, ITS를 새 하드웨어로 옮기는 데 너무 많은 노력이 필요하였다. 이제 VAX에서 작동하는 베클리 계열 유닉스 변종이 가장 좋은 해킹 시스템이 되었으며, 미래에 대한 식견을 가진 사람이라면 마이크로컴퓨터의 놀라운 성장력이 모든 것을 밀어낼 것으로 예측했다.

이 무렵 레비Levy는 ‘해커’라는 책을 썼다. 그의 중요한 정보 제공자 중 한 명은 MIT 인공지능 연구소의 대표 인물이자 연구소 기술의 상업화에 가장 완고하게 타협을 거부한 리처드 스톤먼Richard M. Stallman(Emacs 개발자)이었다.

스톤먼(보통 이름 약칭과 로그인 이름인 RMS로 알려진)은 자유 소프트웨어 재단을 만들고 고품질의 자유 소프트웨어를 만드는 일에 몸을 바쳤다. 레비는 그를 ‘진정한 마지막 해커’로 칭송하였지만, 다행히도 잘못된 칭송이었다.

스톤먼의 위대한 계획은 80년대 초반에 겪던 해커 문화의 변동을 깔끔하게 정리하였다. 1982년 그는 C로 작성해 자유롭게 이용할 수 있는 유닉스의 완전한 복제본을 만들기 시작하였다. 따라서 ITS의 정신과 전통은 더 새로운 유닉스와 VAX 중심 해커 문화의 중요한 일부로 보존되었다.

이 무렵 마이크로 칩과 근거리 네트워크LAN 기술이 해커 문화에 중대한 영향을 미치기 시작하였다. 이더넷과 모토로라 68000 마이크로 칩은 그 잠재성에 바탕을 둔

강력한 결합체로 등장하였으며, 우리가 현재 워크스테이션이라 부르는 것의 첫 번째 세대를 만들어내기 위해 여러 다른 시도가 시작되었다.

1982년 버클리의 유닉스 해커 그룹이 비교적 저렴한 68000 기반 하드웨어에서 동작하는 유닉스가 폭넓은 애플리케이션에서 우세한 결합임을 증명할 것이라는 믿음으로 썬 마이크로시스템Sun Microsystems을 창립하였다. 그들은 옳았고, 그들의 전망은 전체 산업의 형태를 만들어냈다. 워크스테이션은 아직 누구나 구입할 수 있는 정도는 아니었지만 회사와 대학에서 구입하기에는 저렴하였다. 이들의 네트워크(사용자에 대한)는 급속하게 이전 VAX와 다른 시분할 시스템을 교체하였다.

## 상용 유닉스 시대

1984년, 즉 AT&T가 해체되고 유닉스가 최초로 상업 제품이 되었을 때 해커 문화에서 가장 중요한 구분선은 비교적 응집력 있는 인터넷과 유즈넷 중심의 '네트워크 국가(그리고 대부분 유닉스를 운영하는 미니컴퓨터 또는 워크스테이션급의 기계를 사용하는)'와 마이크로컴퓨터 광신도의 거대한 오지 사이였다.

썬Sun과 다른 회사가 만들어낸 워크스테이션급 기계들은 해커들에게 새로운 세계를 열어 주었다. 해당 기계들은 고성능 그래픽이 가능하고 네트워크를 통해 공유 데이터를 전송할 수 있도록 만들어졌다. 1980년대에 해커 문화는 이러한 기능에서 가장 많은 것을 얻어낼 수 있는 소프트웨어와 도구를 만드는 도전에 심취해 있었다. 버클리 유닉스는 ARPAnet 프로토콜의 내장 지원을 개발하였으며, 이는 네트워킹 문제에 대한 해결책을 제공하고 이후 인터넷의 성장을 뒷받침하였다.

워크스테이션 그래픽을 제어하려는 여러 가지 시도가 있었다. 널리 보급된 것의 하나는 X 윈도우 시스템이었다. 성공의 중요 요소는 X 윈도우 개발자들이 해커 윤리에 따라 기꺼이 소스를 자유롭게 공개하여 인터넷을 통해 보급될 수 있도록 한 것 이었다. 독점적인 그래픽 시스템(썬이 제공한 것을 포함하여)에 대한 X 윈도우의

승리는 몇 년 후 유닉스 자체에 심대한 영향을 미친 변화에 대한 전조였다. 또한 종종 ITS/유닉스 경쟁에서 약간의 파를 가르는 성격의 잡음이 있었다(대부분은 이전 ITS 사용자 측에서). 그러나 마지막 ITS 기계는 1990년에 영원히 사라지게 되었다. 열성자들은 더 이상 설 자리가 없었다. 그러나 대부분은 불만이 있었지만 유닉스 문화로 동화되어 갔다.

네트워킹이 가능한 해커 문화 안에서 1980년대의 큰 경쟁은 버클리 유닉스와 AT&T 버전 사이의 열성자들에 의해 생겨났다. 때때로 여러분은 아직도 그 당시의 포스터를 찾을 수 있는데, 스타워즈 영화의 X 왕 전투기가 AT&T 로고가 그려진 폭발하는 데스스타에서 빠져나오는 그림이다. 버클리 해커들은 자신을 비정한 회사 제국과 싸우는 반란군으로 간주하고 싶어했다. AT&T 유닉스는 시장에서 BSD/Sun을 따라오지 못했지만 표준 전쟁에서는 승리하였다. 1990년이 되자 AT&T와 BSD 버전은 구분하기가 어려워졌고 서로의 혁신을 많이 채용하였다.

1990년대가 열리자 80년대의 워크스테이션 기술은 새롭고, 저가이며, 고성능의 인텔 386 칩과 그 후속작에 기반을 둔 개인용 컴퓨터에 의해 분명히 위협받는 것처럼 보이기 시작하였다. 처음으로 해커들은 각자 10년 전 미니컴퓨터의 성능과 저장 능력에 맞먹는 개인용 컴퓨터를 가질 수 있었다. 바로 전체 개발 환경을 지원하고 인터넷과 통신할 수 있는 유닉스 엔진이다.

MS-DOS 세계는 이러한 현상을 간과한 채, 나름의 영역에 만족했다. 이러한 초기의 마이크로컴퓨터 열성자들이 빠르게 DOS와 맥<sup>Mac</sup> 해커의 인기를 ‘네트워크 국가’ 문화의 그것보다 훨씬 크게 늘려 갔지만 자각하는 문화 자체를 이루지는 못하였다. 변화의 정도는 너무나 빨라서 50여 개의 서로 다른 기술적 문화가 하루살이처럼 태어나고 사라져 갔다. 그러나 그들은 은어, 전승, 신화적 이야기의 일반적인 전통을 개발하는 데 필요한 안정성을 획득하지는 못하였다. 즉, UUCP나 인터넷에 비교될 만한 정말 주도적인 네트워크의 부재는 그 자체가 네트워크 국가가 되지 못

하게 하였다. CompuServe나 Genie와 같은 상용 온라인 서비스에 대한 폭넓은 접근이 뿌리내리려 했지만 개발 도구가 포함되지 않은 비 유닉스 운영체제는 소스가 거의 돌아다니지 못한다는 사실을 의미하였다. 따라서 협력 해킹의 전통이 만들 어지지 못하였다.

인터넷에 의해 (비)조직화되고 이제 대략 유닉스 기술 문화로 구분할 수 있는 해커 문화의 주류는 상업 서비스에 대해서는 신경 쓰지 않았다. 그들은 더 좋은 도구와 인터넷을 더욱 많이 원했고 저렴한 32비트 PC는 그 두 가지를 모든 이에게 약속하였다.

그러나 소프트웨어는 어디에 있는가? 상용 유닉스는 수천 달러의 가격을 형성했으므로 여전히 비쌌다. 1990년 초반에 여러 회사가 PC급 기계에 이식된 AT&T나 BSD 유닉스를 판매하였다. 성공은 잘 모르겠지만 가격은 크게 내려가지 않았고 (가장 나쁜 일은) 수정하고 재배포할 수 있는 운영체제의 소스를 얻을 수 없었다. 전통적인 소프트웨어 비즈니스 모델은 해커들이 원하는 것을 주지 않았던 것이다.

자유 소프트웨어 재단도 마찬가지였다. RMS가 오래전에 약속한 해커를 위한 유닉스 커널인 HURD의 개발은 수 년간 지체되었으며, 1996년까지도 사용 가능한 커널을 만드는 데 실패하였다(1990년 정도에 FSF는 유닉스와 비슷한 운영체제의 다른 어려운 부분을 거의 모두 제공하긴 했다).

설상가상으로 1980년 초반까지 독점적인 유닉스를 상업화하기 위한 10여 년간의 노력이 실패로 끝났다는 사실이 확실해지고 있었다. 유닉스의 플랫폼 간 호환성에 대한 약속은 여러 가지 상용 유닉스 버전 사이의 논쟁 속에 잊혀졌다. 즉, 상용 유닉스 업체들의 답답하고, 맹목적이며, 부적절한 마케팅 때문에 놀랍도록 열등한 윈도우 운영체제의 기술로도 시장의 많은 부분을 점유할 수 있게 했다. 1993년 초에 적대적인 관측자들은 유닉스 이야기는 거의 끝났다고 생각할만한 근거를 가졌고 해커 부족의 운명도 그럴 것이었다. 그리고 컴퓨터 업계 언론에서는 적대적인 관측

자들이 계속 있어 대부분은 1970년대 후반부터 6개월 간격으로 계속 유닉스가 죽음이 임박했음을 예측했다.

이 시대에는 개개인의 기술 영웅주의는 끝났으며, 소프트웨어 산업과 초기의 인터넷은 점점 더 마이크로소프트와 같은 거인에 의해 지배된다는 것이 일반적인 믿음이었다. 유닉스 해커의 첫 세대는 늙고 지쳐 보였다(버클리의 전산 과학 연구 그룹 CSRG은 1994년에 활력을 잃고 해체되었다). 침울한 시대였다.

다행히 언론 및 대부분의 해커에게도 눈에 띄지 않게 진행되는 일이 있어 1993년 후반과 1994년에 인정받을 만한 놀라운 제품을 만들어냈다. 결국 이 일은 해커 문화를 아주 다른 방향으로 이끌어 생각지도 못했던 성공을 거두었다.

## 초기의 공개 유닉스

HURD의 실패가 남겨둔 틈새 사이로 리누스 토르발스 Linus Tovalds라는 헬싱키 대학생이 들어왔다. 1991년에 그는 프리 소프트웨어 재단의 도구를 사용하여 386 기계를 위한 공개 유닉스 커널을 개발하기 시작하였다. 초기의 연속적인 성공은 많은 인터넷 해커들을 끌어들여 리눅스가 완전히 자유롭고 재배포 가능한 소스로 구성된 제대로 된 유닉스인 리눅스를 개발하도록 도왔다.

리눅스에게는 경쟁자가 없지 않았다. 1991년 리누스 토르발스의 초기 실험과 동시에 윌리엄 William과 라인 줄리츠 Lynne Jolitz는 실험적으로 386에 BSD 유닉스 소스를 이식하였다. BSD 기술과 리눅스의 미숙한 초기 노력을 비교하던 대부분의 관찰자는 BSD 포트 port가 PC의 가장 중요한 공개 유닉스가 될 것으로 예측하였다. 그러나 리눅스의 가장 중요한 특징은 기술적인 것이 아니라 사회적인 것이었다. 리눅스의 개발 이전에는 모두가 운영체제처럼 복잡한 소프트웨어는 비교적 적고 잘 구성된 그룹에 의해 주의 깊게 조정되어 개발되어야 한다고 믿고 있었다. 이 모델은 상업 소프트웨어와 1980년대에 자유 소프트웨어 재단이 만들어 낸 위대한 프리웨

여 대성당에는 전형적인 방식이었고 현재에도 여전히 일반적인 방식이다. 또한 졸리츠의 오리지널 386 BSD 포트에서 나온 FreeBSD/NetBSD/Open BSD 프로젝트에도 그랬다.

리눅스는 완전히 다른 방식으로 발달하였다. 거의 처음부터 인터넷에 의해서만 조정되는 수많은 자원자에 의해 우연히 해킹되었다. 품질은 엄격한 표준이나 독재에 의해 이루어지지 않았으며, 매주 릴리즈되면 며칠 안에 수백 명의 사용자로부터 피드백을 받고, 개발자들이 지속해서 변화를 소개하는, 다원주의적인 적자생존의 선택을 할 수 있게 하는 그런 단순한 전략에 의해서 관리되었다.

1993년 후반 리눅스는 안정성과 신뢰성에서 많은 상용 유닉스와 경쟁할 수 있었으며 아주 많은 소프트웨어를 운영하였다. 심지어 상용 애플리케이션 소프트웨어의 이식도 이끌어내기 시작하였다. 리눅스 개발의 간접 효과는 판매할 개발자와 해커들이 없는 대부분의 조그만 상업 유닉스 업체들을 전멸시키는 것이었다. 소수의 생존자 중 하나인 BSDI(버클리 시스템 디자인사)는 자사의 BSD 기반 유닉스에 전체 소스를 제공하고 해커 사회와의 가까운 관계 유지에 노력하여 성공하였다. 이러한 개발 이야기는 당시 해커 문화 내에서도 많이 회자되지 못하였고 밖에서는 전혀 이야기가 없었다. 종말에 대한 반복적인 예측을 무시하던 해커 문화는 자신의 이미지대로 상업 소프트웨어 세계를 다시 만들어내려 했다. 그러나 이런 경향이 명백해지는 데는 5년 이상이 걸렸다.

## 웹의 폭발적인 성장

리눅스의 초기 성장은 또 다른 현상과 동반 상승하였다. 바로 일반인들이 인터넷을 접하기 시작한 것이다. 1990년 초반, 한 달에 몇 달러로 일반인에게 인터넷 연결을 제공하는 인터넷 서비스 제공 산업이 번성하기 시작했다. 월드 와이드 웹의 발명 이후, 이미 발전 단계에 있던 인터넷은 위험천만할 정도로 발전에 가속이 붙었다.

1994년 버클리의 유닉스 개발 그룹이 공식적으로 해체된 해에 서로 다른 공개 유닉스 버전(리눅스와 386 BSD의 후손들)은 해킹 활동에 중추적 역할을 하였다. 리눅스는 CD-ROM에 담겨 상업적으로 배포되어 날개돋친 듯 팔리고 있었다. 1995년 후반에 주요 컴퓨터 회사들은 자신의 소프트웨어와 하드웨어가 인터넷에 적합하다고 선전하는 그럴듯한 광고를 내기 시작하였다!

1990년 후반에 해커 문화의 주요 활동은 리눅스 개발과 인터넷의 주류화였다. 월드 와이드 웹은 마침내 인터넷을 대중 매체로 만들었고, 1980년대와 1990년 초반의 많은 해커는 대중에게 인터넷 접속 서비스를 판매하거나 제공하는 ISP<sup>Internet Service Providers</sup>를 시작하였다.

인터넷의 고속 성장은 해커 문화에 중추적 지위와 정치적 영향력을 구가할 수 있는 계기를 마련해 주었다. 1994년과 1995년 해커 행동주의는 정부 제어 아래의 강력한 암호화 기법을 갖게 하자는 Clipper 제안서의 발현을 저지했다. 1996년 해커들은 잘못 이름 붙여진 ‘통신 품위법’<sup>Communications Decency Act, CDA</sup>을 무효로 하기 위한 폭넓은 활동을 결집하여 인터넷의 검열을 방해하였다. 이러한 활동 덕분에 우리는 역사에서 벗어나 현재에 들어섰다. 또한 여러분의 역사가들이 관찰자가 아닌 행동가가 되는 시대에 들어섰다. 이 이야기는 ‘해커들의 반란’에서 계속될 것이다.

모든 정부는 작든 크든 민중과 대립하는 존재의 조합이다. 덕이 없기로는 지배자나 피지배자나 다를 것이 없다. 정부의 힘은 자신과 동일한, 민중의 침착한 감정과 힘의 과시에 의한 정해진 범위 안에서만 유지될 수 있다.

— 벤저민 프랭클린 바쉬(Benjamin Franklin Bache),  
필라델피아 오로라(Philadelphia Aurora)의 사설에서, 1794

## 2 | 버클리 유닉스의 20년

AT&T 소유에서 자유로운 재배포가 가능하기까지

마샬 커크 맥퀴식 Marshall Kirk McKusick

최준호 역

### 초기 역사

肯 톰슨Ken Thompson과 데니스 리치Dennis Ritchie는 1973년 11월에 퍼듀Purdue 대학에서 있었던 SOSP<sup>Symposium of Operating Systems Principles</sup>에서 최초의 유닉스 논문을 발표하였다. 캘리포니아 버클리 대학의 밥 패브리Bob Fabry 교수는 당시 참석 중이었는데, 버클리에서 실험해보기 위해 시스템의 복사본을 얻고 싶어했다.

그 당시 버클리에는 배치batch 처리만 하는 커다란 메인프레임 시스템 하나만 있었으므로 먼저 버전 4의 유닉스를 실행하기에 적합한 PDP-11/45 시스템을 갖추어야 했다. 버클리의 전산과학과는 수학과, 통계학과와 함께 PDP-11/45를 공동으로 구입할 수 있었다. 이러한 과정을 통해 드디어 1974년 1월 버전 4 테이프가 도착하였고, 대학원생인 키스 스탠디포드Keith Standiford가 유닉스를 설치하였다.

퍼듀 대학의 톰슨은 당시 대부분의 시스템처럼 버클리에 설치된 시스템에는 관여하고 있지 않았다. 그렇지만 몇 가지 이상한 시스템 충돌이 일어나면서 그의 전문지식이 필요하게 되었으며, 이 때문에 연구에 참여하게 되었다. 버클리에는 자동응답 기능이 없는 300보오의 음성 모뎀밖에 없었으므로 톰슨은 기계실에 있는 스탠디포드에게 전화를 하여 전화를 모뎀에 삽입하도록 하였다. 이런 방법으로 톰슨은 뉴저지에서 크래쉬덤프를 원격으로 디버깅할 수 있었다.

대부분의 층돌은 문서에 있는 것과는 달리 디스크 컨트롤러가 중첩된 탐색을 신뢰성 있게 수행하지 못했기 때문에 일어났다. 톰슨이 보게 된 베클리의 11/45는 같은 컨트롤러에 두 개의 디스크가 달린 최초의 시스템 중 하나였다. 톰슨의 원격 디버깅은 베클리와 벨 연구소 사이에 이루어진 최초의 협동 작업이었다. 베클리와 작업을 공유하려 했던 벨 연구소 연구원들의 자발성은 베클리에서 해당 소프트웨어를 사용할 수 있도록 신속하게 개선하는 데 도움이 되었다.

유닉스는 곧 안정적으로 실행되었지만 전산과학, 수학, 통계학과의 연합은 몇 가지 문제에 직면하게 되었다. 수학과와 통계학과는 DEC의 RSTS 시스템을 실행하고 싶어했던 것이다. 많은 토론 뒤에 각 과는 8시간씩 나누어 사용한다는 합의에 이르게 되었다. 유닉스는 8시간, RSTS는 16시간 사용한다는 것이었다. 공정성을 기하기 위해 시간 배분은 매일 순환하도록 하였다. 따라서 유닉스가 어느 날 오전 8시에서 오후 4시까지 실행되었다면 다음 날은 오후 4시에서 자정까지, 그 다음 날은 자정부터 오전 8시까지 실행되는 식이었다. 이상한 스케줄이었지만 운영체제 수업을 듣는 학생은 배치 기계보다는 유닉스에서 프로젝트를 하는 것을 더 좋아했다.

유진 왕Eugene Wong과 마이클 스톤브레이커Michael Stonebraker 교수는 둘 다 배치 환경의 제한에 불편해했다. 그래서 그들이 이끄는 INGRES 데이터베이스 프로젝트 그룹은 배치 기계에서 유닉스가 제공하는 대화형 환경으로 이전한 첫 번째 그룹 중 하나가 되었다. 그들은 11/45에서의 모자란 시간과 이상한 시간대를 참을 수 없게 되었다. 그래서 1974년 봄에 새로이 베전 5가 실행되는 11/40을 구입하였다. 1974년 가을 INGRES의 첫 번째 배포판으로 INGRES 프로젝트는 소프트웨어를 배포하는 전산과학과의 첫 번째 그룹이 되었다. 이후 6년간 수백 개의 INGRES 테이프가 배송되었으며 실제 시스템을 설계하고 만드는 베클리의 명성을 확립하는데 도움을 주었다.

INGRES 프로젝트가 11/45에서 떠났지만 남은 학생들에게 주어진 사용 가능 시간은 여전히 부족했다. 열악한 환경을 개선하기 위해 스톤브레이커와 패브리 교수는 1974년에 전산과학과를 위한 교육용 11/45 두 개를 더 구입하기로 결정했다. 1975년 초에 비용이 마련되었다. 이와 거의 동시에 DEC은 11/70을 발표하였는데, 이는 11/45보다 매우 뛰어나 보였다. 그래서 11/45 두 개를 사기 위한 비용은 11/70 한 대를 사는 데 모였으며 시스템은 1975년 가을에 도착하였다. 11/70의 도착과 함께 톰슨은 모교인 버클리 대학교에서 1년간 방문 교수로 안식년을 보내 기로 하였다. 톰슨은 제프 스크리브만 Jeff Schriebman, 그리고밥 크리들 Bob Kridle과 함께 새롭게 설치된 11/70에 최신의 유닉스인 버전 6을 이식하였다.

1975년 가을 거의 남의 눈에 띄지 않는 대학원생이었던 빌 조이 Bill Joy와 척 헬리 Chuck Haley가 버클리에 도착하였다. 이들은 곧 새로운 시스템에 흥미를 갖게 되었다. 처음에 이들은 11/70 기계실을 배회하며 톰슨이 해킹하던 파스칼 시스템을 같이 해킹하기 시작했다. 이들이 확장하고 개선한 파스칼 인터프리터는 많은 학생이 선택하는 프로그래밍 시스템이 될 정도였는데 이는 뛰어난 오류 복구 방식, 빠른 컴파일과 실행 시간 때문이었다.

Model 33 텔레 타입을 ADM-3 화면 터미널로 바꾸면서 조이와 헬리는 ed 편집기의 제약에 불편함을 느꼈다. 그래서 런던 퀸 메리 Queen Mary 대학의 조지 쿨루리스 George Coulouris 교수에게서 얻은 em이라는 편집기에 기반을 둔 행단위 편집기인 ex를 만들었다.

1976년 여름이 끝나갈 무렵에 톰슨이 떠나자 조이와 헬리는 유닉스 커널의 내부를 탐험하는 데 흥미를 갖기 시작하였다. 스크리브만이 주의 깊게 지켜보는 가운데, 이들은 먼저 벨 연구소의 ‘50개의 변경 사항 fifty changes’ 테이프에서 제공되던 수정본과 향상된 유닉스를 먼저 설치하였다. 소스 코드를 살펴보는 방법을 터득한 뒤 이들은 커널 병목 현상을 해결하기 위해 여러 개선점을 제안하기도 했다.

## 초기 배포판

그동안 파스칼 컴파일러의 오류 복구 동작에 대한 관심으로 사람들은 시스템의 복사본을 요청하였다. 1977년 초, 조이는 ‘버클리 소프트웨어 배포판’<sup>Berkeley Software Distribution</sup>을 만들어냈다. 이 첫 번째 배포판은 파스칼 시스템, 그리고 파스칼 소스 코드 안의 잘 드러나지 않는 서브 디렉터리에 편집기 ex가 들어 있었다. 다음 해 조이는 배포판 사무를 보며 시스템의 복사본 30여 개를 발송하였다.

화면에 주소 부여가 가능한 커서를 제공하는 일부 ADM-3a 터미널이 도착하자 조이는 결국 버클리에 화면 기반 편집을 가능하도록 한 vi를 작성할 수 있었다. 그러나 그는 곧 곤경에 처하게 되었다. 대학에 재정이 부족한 것은 자주 있는 일이므로 이전 장비가 한 번에 교체되는 일은 거의 없었다. 그래서 그는 여러 가지 다른 터미널의 갱신을 최적화하는 지원 코드보다 화면을 다시 그리는 조그만 인터프리터를 사용하여 화면 관리를 통합하고자 했다. 이 인터프리터는 터미널의 특성을 기술함으로써 파생되었으며, 이런 노력은 결과적으로 termcap이 되었다.

1978년 중반 소프트웨어 배포판을 갱신할 필요가 생겼다. 파스칼 시스템은 늘어나는 사용자의 피드백을 통해 놀라울 정도로 안정되었으며, 두 단계를 거치도록 분리되어 PDP-11/34에서도 실행될 수 있었다. 갱신 결과는 ‘두 번째 버클리 소프트웨어 배포판’이었으며, 곧 2BSD라는 약어로 쓰이게 되었다. 향상된 파스칼 시스템, vi와 여러 터미널을 위한 termcap이 포함되었다. 또다시 빌 조이는 혼자서 배포판을 구성하였고, 전화에 응답하고, 사용자의 피드백을 시스템으로 통합하였다. 다음 한 해 동안 거의 75개의 테이프가 발송되었다. 조이는 그 다음 해 다른 프로젝트로 옮겼지만 2BSD 배포판은 계속 늘어났다. 이 배포판의 최종 버전인 2.11BSD는 아직도 전 세계의 여러 곳에서 운영되는 수백 개의 PDP-11에 사용되는 완전한 시스템이다.

## VAX 유닉스

1978년 초, 리처드 페이트만Richard Fateman 교수는 맥시마Macsyma에 관한 작업을 계속하기 위해(원래는 PDP-10에서 시작하였다) 더 큰 주소 공간을 갖는 기계를 찾기 시작하였다. 새로이 발표된 VAX 11/780은 그런 요구 사항을 만족하였고 예산 내에서 구입할 수 있었다. 페이트만과 다른 13명의 교수진은 VAX를 구입하기 위해 과 기금의 일부를 더하여 NSF에 제안서를 작성하였다.

처음에 VAX는 DEC의 VMS 운영체제를 실행하였지만, 전산과학과는 유닉스 환경을 사용해왔으며 계속 사용하고 싶어하였다. 그래서 VAX가 도착한 뒤 얼마 있지 않아 페이트만은 벨 연구소의 존 리저John Reiser와 톰 런던Tom London이 만든 VAX용 유닉스 32/V 포트의 복사본을 마련하였다. 32/V가 버전 7 유닉스 환경을 VAX에서 제공하기는 하지만 VAX 하드웨어의 가상 메모리 기능의 장점을 살리지는 않았다. 이는 과거의 다른 PDP-11처럼 완전히 스왑 기반 시스템이었다. 즉, 버클리의 맥시마 그룹에게 가상 메모리 부족은 프로세스 주소 공간이 새 VAX에서 기본적으로 1MB던 물리적 메모리의 크기에 제한된다는 사실을 의미하였다.

이 문제를 해결하고자 페이트만은 버클리 시스템 학과의 도미니코 페라리Domenico Ferrari 교수에게 그의 그룹이 유닉스를 위한 가상 메모리를 작성할 수 있는지의 가능성을 알아봐 줄 것을 의뢰했다. 페라리의 학생인 오잘프 바바오관루Ozalp Babaoglu는 VAX에서 워킹 셋 페이징 시스템을 구현하는 방법을 찾아보기 시작하였다. 이 작업은 VAX에 참조 비트가 없었기 때문에 상당히 복잡하였다. 바바오관루가 첫 번째 구현물을 완성해 갈 무렵, 그는 빌 조이에게 유닉스 커널의 복잡성을 이해하기 위한 도움을 청했다. 바바오관루의 접근법에 호기심이 생긴 조이는 이 코드를 32/V에 이식하고 디버깅하는 일을 같이 도왔다.

불행히도 버클리에는 시스템 개발과 일반적인 사용을 위한 VAX가 한 대밖에 없었다. 그래서 크리스마스 주간을 포함한 몇 주 동안 인내심 좋은 사용자들은 32/V

와 ‘가상 VAX/유닉스’를 번갈아가며 로그인하게 되었다. 후자 시스템에서의 작업에서는 종종 뜻밖의 중지 후 몇 분 뒤에 32/V 로그인 프롬프트가 나오는 일이 있었다. 1979년 1월 대부분의 버그가 수정되었고 32/V는 역사 속으로 사라졌다.

조이는 32비트 VAX가 곧 16비트 PDP-11을 쓸모없게 만들 것이라는 사실을 알았으며 2BSD 소프트웨어를 VAX로 이식하기 시작하였다. 피터 케슬러Peter Kessler 와 내가 파스칼 시스템을 이식하고, 조이는 ex, vi, C 쉘, 2BSD 배포판의 작고 수 많은 유ти리티를 이식하였다. 1979년 말 완전한 배포판이 구성되었다. 이 배포판은 가상 메모리 커널, 표준 32/V 유ти리티, 2BSD에서의 추가 사항을 반영했다. 1979년 12월 조이는 버클리에서 나온 최초의 VAX 배포판인 3BSD의 100여 개 복사본을 먼저 발송하였다.

벨 연구소의 마지막 릴리즈는 32/V였다. 이후 시스템 III와 이후의 시스템 V인 AT&T의 모든 유닉스 릴리즈는 안정적인 상업적 릴리즈를 강조하는 다른 그룹이 관리하였다. 유닉스의 상업화와 함께 벨 연구소의 연구원들은 더 이상 진행 중인 유닉스 연구의 약전 병원 역할을 할 수 없었다. 하지만 연구자들이 유닉스 시스템을 계속 수정해 나감에 따라 연구용 릴리즈를 만들어야 할 기구가 필요하게 되었다. 버클리는 유닉스에 초기부터 관여했고 유닉스 기반 도구를 릴리즈했던 역사 때문에 이전에 벨 연구소에서 제공했던 역할을 빠르게 대신하게 되었다.

## DARPA의 지원

그 동안에 DARPA Defense Advanced Research Projects Agency, 고등 방위 연구 프로젝트국 기획관의 사무실에서는 버클리의 작업에 큰 영향을 미치게 될 토론이 벌어지고 있었다. DARPA의 초기 성공은 모든 주요한 연구 센터를 하나로 연결하는 국가적인 컴퓨터 네트워크를 구성하는 것이었다. 그 당시 토론에 참여한 사람들은 이런 센터의 많은 컴퓨터가 수명이 다했고 교체해야 한다는 사실을 알고 있었다. 가장 큰 교체 비용은 연구용 소프트웨어를 새로운 기계에 이식하는 것이었다. 게다가 많은 사이

트는 하드웨어와 운영체제의 다양성 때문에 소프트웨어를 공유할 수 없었다. 그런데 하나의 하드웨어 벤더를 선택하는 것은 연구 그룹의 폭넓은 여러 가지 컴퓨팅 요구 사항과 단일 제조 회사에 의존하는 것이 바람직스럽지 않다는 이유로 실용적이지 않았다. 따라서 DARPA의 기획자들은 가장 좋은 해결책으로 컴퓨터가 아닌 운영체제 수준에서 네트워크를 통합하는 것이라고 결정하였다. 많은 토론을 거친 끝에 안정성이 증명된 유닉스가 표준으로 선택되었다.

1979년 가을, 밥 패브리는 버클리가 DARPA 사람들이 사용하도록 3BSD의 향상된 버전을 개발한다는 제안서를 작성하여 DARPA가 가진 유닉스에 대한 관심에 응답하였다. 패브리는 그의 제안서 복사본을 DARPA 이미지 프로세싱과 VLSI 계약자, ARPAnet의 개발자인 BBNC Bolt Beranek and Newman사의 대표들에게 제공하였다. 버클리가 동작하는 시스템을 만들 수 있는지에 대해서는 약간의 우려가 있었으나 1979년 12월의 3BSD 릴리즈는 그런 의심을 떨쳐버렸다.

패브리의 주장을 확인해 준 3BSD 릴리즈의 명성이 알려짐에 따라 그는 DARPA 와 1980년 4월에 시작하는 18개월짜리 계약을 할 수 있었다. 이 계약은 DARPA 계약자들이 필요한 기능을 추가하는 것이었다. 이 계약의 후원으로 밥 패브리는 CSRG<sup>Computer Systems Research Group</sup>, 컴퓨터 시스템 연구회라는 기구를 만들었다. 그는 곧 로라 퉰Laura Tong을 고용하여 프로젝트 관리를 하도록 하였다. 이후 패브리는 소프트웨어 개발을 관리할 프로젝트 리더를 찾는 데 관심을 돌렸다. 패브리는 조이가 막 박사 자격시험을 통과했기 때문에 소프트웨어 개발직보다는 학위 논문을 완성하는 데 집중할 것으로 생각했다. 하지만 조이는 다른 계획이 있었다. 3월 초 어느 날 그는 패브리에게 전화를 걸어 유닉스의 차후 개발을 맡는 데 관심이 있음을 알려주었다. 제안에 놀라기는 했지만 패브리는 곧 동의하였다.

프로젝트는 곧 시작되었다. 로라 퉰은 조이의 이전 배포판보다 더 많은 양의 주문을 다룰 수 있는 배포 시스템을 만들었다. 패브리는 AT&T의 밥 거피Bob Guffy와 베

클리 대학의 변호사 모두가 납득할 수 있는 조항 아래에서 유닉스를 공식적으로 릴리즈하도록 조정하였다. 조이는 짐 컬프 Jim Kulp의 작업 제어 job control, 자동 리부트, 1K 블록 파일 시스템, 최신 VAX 11/750 기계의 지원을 통합하였다. 1980년 10월, 파스칼 컴파일러, Franz 리스프 시스템, 향상된 이메일 처리 프로그램을 포함하는 정리된 배포판이 4BSD로 릴리즈되었다. 9달 동안 약 150개의 복사본이 발송되었다. 라이선스는 기계 단위보다는 기관 단위였다. 따라서 이 배포판은 약 500개의 기계에서 실행되었다.

버클리 유닉스 배포판이 널리 퍼짐에 따라 여러 가지 비평이 나타나기 시작했다. 스탠퍼드 연구소 Stanford Research Institute의 데이비드 카쉬탄 David Kashtan은 VMS와 버클리 유닉스에서 실행한 벤치마크의 결과를 설명한 논문을 작성하였다. 이 벤치마크는 VAX용 유닉스 시스템에 심각한 성능 문제가 있다는 사실을 보여주었다. 몇 달 동안 자신의 장래 계획을 제쳐 두고 조이는 커널을 시스템 수준에서 조율하기 시작하였다. 몇 주 후 그는 카쉬탄의 벤치마크가 유닉스에서도 VMS만큼 실행되도록 만들어질 수 있다는 반증용 논문을 작성하였다.

계속 4BSD를 발송하는 대신 로버트 엘즈 Robert Elz의 자동 설정 코드가 추가되고 튜닝된 시스템은 1981년 6월에 4.1BSD로 릴리즈되었다. 약 2년간 400여 배포판이 발송되었다. 본래의 의도는 5BSD 릴리즈를 만드는 것이었지만 AT&T가 자신의 상업용 유닉스 릴리즈인 시스템 V와 버클리의 5BSD라 붙여진 이름은 고객들에게 혼란만 안겨줄 수 있다는 이유로 반대하였다. 따라서 이 문제를 해결하기 위해 버클리는 계속 4BSD를 유지하고 부 버전 번호만 올린다는 이름 규칙에 동의하였다.

## 4.2BSD

4.1BSD의 릴리즈와 함께 성능에 대한 여러 가지 불만은 없어졌다. DARPA는 버클리와 첫 번째 계약의 결과에 충분히 만족하고 이전보다 다섯 배를 더 투자하는 2년간의 새로운 계약을 맺었다. 자금의 반은 유닉스 프로젝트에게, 다른 반은 전산

과학과의 다른 교수들에게 돌아갔다. 계약에 따라 시스템에 대한 중요 작업이 이루어졌고, DARPA의 연구자들은 자신의 작업을 더 잘할 수 있었다.

DARPA의 요구에 따라 목표가 정해지고 시스템의 변경 사항을 규정하기 위한 작업이 시작되었다. 특히 새 시스템의 방향은 다음과 같았다. 이용할 수 있는 디스크 테크놀러지의 속도에 따라 성능을 높일 수 있는 더 빠른 파일 시스템을 포함하고, 수 GB의 주소 공간 요구 사항을 만족하는 프로세스를 지원하고, 연구자들이 분산 시스템에서 작업할 수 있도록 하는 유연한 프로세스 간 통신 기능<sup>Inter Process Communication, IPC</sup>을 제공하고, 네트워킹 지원을 통합하여 새 시스템을 운영하는 기계가 쉽게 ARPAnet에 참여할 수 있도록 할 예정이었다.

새 시스템을 규정하는 작업을 돋기 위해 DARPA에 있는 베클리와의 계약 감독관인 듀에인 아담스<sup>Duane Adams</sup>는 설계 작업을 돋고 연구자들의 요구가 정리되도록 ‘운영 위원회’로 알려진 그룹을 조직하였다. 이 위원회는 1981년 4월에서 1983년 6월까지 1년에 두 번 회의를 하였다. 여기에 속한 사람은 베클리 대학교의 밥 패브리, 빌 조이, 그리고 샘 레플러와 BBN<sup>Bolt Beranek and Newman</sup>의 앤린 네메스, 롭 구르위츠, 벨 연구소의 데니스 리치, 스탠퍼드 대학의 키스 랜츠, 카네기 멜론 대학의 릭 라쉬드, MIT의 버트 할스테드, ISI<sup>Information Science Institute</sup>의 댄 런치, DARPA의 듀에인 애덤스와 밥 베이커, UCLA의 제리 폼페이었다. 1984년부터 이 회의는 워크숍으로 바뀌어 더 많은 사람을 수용하게 되었다.

새 시스템에 포함될 기능을 제안하는 첫 번째 문서는 1981년 7월 운영 위원회와 베클리 외의 다른 사람이 돌려 보았고 많은 긴 토론이 있었다. 1981년 여름, 나는 CSRG에 속하여 새 파일 시스템 구현 작업을 맡게 되었다. 여름 내내 조이는 프로세스 간 통신 기능의 프로토타입 버전을 구현하는 데 집중하였다. 1981년 가을, 샘 레플러는 빌 조이와 함께 일하기 위해 CSRG의 상근 직원으로 합류하였다.

롭 구르위츠 Rob Gurwitz가 버클리에 TCP/IP 프로토콜의 초기 구현을 릴리즈했을 때 조이는 이것을 시스템에 통합하고 성능을 조율하였다. 이 작업 중 조이와 레플러에게는 새 시스템이 DARPA 표준 네트워크 프로토콜 이상을 지원해야 할 필요성이 분명해졌다. 따라서 두 사람은 소프트웨어의 내부 구조를 재설계하고, 인터페이스를 세분화하여 여러 개의 네트워크 프로토콜이 동시에 사용될 수 있도록 했다.

내부 구조의 재구성이 끝나고 TCP/IP 프로토콜이 프로토타입 IPC 기능과 함께 통합되었을 때 지역 사용자들이 원격 자원에 접근할 수 있는 기능을 제공하도록 몇 가지 간단한 애플리케이션이 만들어졌다. rcp, rsh, rlogin, rwho와 같은 프로그램은 최종적으로 더 합리적인 기능으로 대체될 임시 도구로 만들어진 것이었다(그래서 눈에 띄는 ‘r’ 접두어를 사용하였다). 4.1a라 불리는 이 시스템은 내부용으로 1982년 4월에 처음 배포되었다. 4.2 릴리즈를 참을성 있게 기다리지 못하는 사이트가 늘어남에 따라 몰래 만든 시스템의 복사본이 퍼지기는 하였지만, 이는 널리 돌려볼 목적으로 만들어진 것이 아니었다.

4.1a 시스템은 완성되기 오래전에 사용하지 않게 되었다. 그러나 사용자들의 피드백은 가치 있는 정보를 제공하였고, 이는 ‘4.2BSD 시스템 매뉴얼’이라는 새 시스템의 정리된 제안서를 만드는 데 사용되었다. 이 문서는 1982년 2월에 만들어졌으며 4.2BSD에서 구현될 시스템 기능이 가져야 할 사용자 인터페이스 제안서의 간결한 설명을 담고 있었다. 또한 4.1a의 개발과 함께 새 파일 시스템의 구현을 완료하였고 1982년 6월까지 4.1a 커널과 완전히 통합하였다. 그 결과는 4.1b라 불리었고 버클리 안의 몇몇 선택된 개발 기계에서만 실행되었다. 조이는 곧 닉칠 시스템에 대한 큰 변화 때문에 교내 배포판도 피하는 것이 좋겠다고 느꼈는데, 특히 4.1a에서 4.1b로 전환하기 위해서는 모든 기계의 파일 시스템을 덤프하고 되돌려놓아야 했기 때문이다. 일단 파일 시스템이 안정되었다고 증명되자 레플러는 새 파일 시스템 관련 시스템 콜을 추가하고, 조이는 프로세스 간 통신 기능을 수정하는 작업을 하였다.

1982년 늦은 봄, 조이는 썬Sun에 입사한다고 알렸다. 여름 내내 그는 썬과 버클리에서 시간을 나누어 보냈다. 그 기간 중 대부분의 시간은 프로세스 간 통신 기능의 수정 사항을 마무리하고 유닉스 커널 소스를 재구성하여 기계 의존성을 독립시키는 일이었다. 조이가 떠나자 레플러가 프로젝트를 완성하는 책임을 지게 되었다. 일부 기한이 이미 지났으며 시스템의 릴리즈는 1983년 봄에 DARPA에게 제출하기로 약속된 상태였다. 주어진 시간 안에 릴리즈를 완성하기 위해 남은 작업을 평가하고 우선권이 지정되었다. 특히 가상 메모리 기능 향상과 프로세스 간 통신 설계의 가장 복잡한 부분에는 낮은 우선순위를 부여하였다(그리고 나중에 완전히 빠지게 되었다). 또한 구현에 1년 이상이 소요되고 유닉스 사용자들의 기대가 높아짐에 따라, 최종 시스템이 완성되기 전까지 사람들을 불잡아 놓을 중간 릴리즈를 구성하기로 결정하였다. 4.1c라 불리는 이 시스템은 1983년 4월에 배포되었다. 많은 벤더는 자신의 하드웨어에 4.2를 이식하기 위한 준비로 이 릴리즈를 사용하였다. 그리고 폴린 슈와츠Pauline Schwartz가 고용되어 4.1c 릴리즈부터 배포 책임을 지게 되었다.

1983년 6월, 밥 패브리Bob Fabry는 CSRG의 관리 제어권을 도미니코 페라리Domenico Ferrari와 수잔 그라함Susan Graham 교수에게 넘기고 4년간의 분주했던 날들로부터 안식의 자유를 얻기 시작했다. 레플러Leffler는 시스템의 완성 작업을 계속하였으며, 새 시그널 기능을 구현하고 네트워킹 지원을 추가하였다. 또한 설치 과정을 단순화하기 위해 독립형 I/O 시스템을 다시 만들었으며, 로버트 엘츠Robert Elz의 디스크 쿼터 기능을 통합하고, 모든 문서를 갱신하고, 4.1c 릴리즈의 버그를 추적하였다. 1983년 8월 이 시스템은 4.2BSD로 발표되었다.

레플러가 4.2를 완성하고 루카스 필름으로 떠난 후 그의 자리는 마이크 카렐스Mike Karels가 맡았다. 카렐스의 2.9BSD PDP-11 소프트웨어 배포판에 대한 이전 경험은 그의 새로운 일자리에 이상적인 배경이 되어 주었다. 1984년 12월, 박사 학위를 얻고 CSRG에 상근으로 마이크 카렐스와 같이 일하게 되었다.

4.2BSD의 인기는 대단했다. 18개월 동안 1,000개 이상의 사이트 라이선스 계약이 이루어졌다. 즉, 4.2BSD는 이전의 모든 베클리 소프트웨어 배포판을 합친 것 보다 더 많이 발송된 것이다. 대부분의 유닉스 벤더는 AT&T의 상용 시스템 V보다 4.2BSD 시스템을 판매하였다. 그 이유는 시스템 V에는 네트워킹도 없고 베클리 패스트 파일 시스템도 없었기 때문이었다. 유닉스의 BSD 릴리즈는 원위치로 돌아가기 전까지 몇 년 동안만 주도적인 상업적 위치에 있었다. 네트워킹과 다른 4.2BSD의 개선 사항이 시스템 V 릴리즈에 통합되자 벤더는 대부분 시스템 V로 돌아갔다. 그러나 나중의 BSD 개발 사항도 시스템 V에 계속 통합되었다.

## 4.3BSD

4.1BSD는 릴리즈와 함께 곧 몇 가지 사항에 대한 비난을 받았다. 대부분의 불평은 시스템이 너무 느리다는 것이었다. 문제의 원인은 새 기능이 튜닝되지 못하여 많은 커널 데이터 구조가 새로운 쓰임새에 잘 맞지 않기 때문이었다. 카렐스와 나는 프로젝트 첫 해를 시스템을 튜닝하고 세련되게 하는 작업으로 보냈다.

시스템을 튜닝하고 네트워킹 코드를 세분화하는 2년간의 작업을 한 후 1985년 6월의 Usenix 콘퍼런스에서 그 해 여름 이후 4.3BSD를 릴리즈 할 것으로 예상한다고 발표하였다. 그러나 우리의 릴리즈 계획은 BBN 사람들에 의해 뜻밖의 중단을 맞이하게 되었다. 그들은 자신들 네트워킹 코드의 최종 버전으로 4.2BSD를 업데이트하지 않았다는 점을 정확하게 지적하였다. 오히려 우리는 여전히 그들이 몇 년 전에 넘겨준 초기 원형을 많이 해킹하여 사용했다. 그들은 BBN이 프로토콜을 구현하고 베클리는 인터페이스를 구현해야 한다고 DARPA에게 호소하였다. 따라서 베클리는 4.3BSD의 TCP/IP 코드를 BBN의 구현으로 교체해야 한다는 것이었다.

마이크 카렐스는 BBN 코드를 얻어 베클리에 원형이 제출된 이후의 작업에 대한 평가를 시작하였다. 그는 가장 좋은 방법은 베클리의 코드 베이스를 교체하지 않고 BBN 코드의 장점만을 베클리 코드 베이스에 통합하는 것이라고 결정하였다. 베클리

리 코드를 유지해야 하는 이유는 4.2BSD가 널리 배포되어 상당한 시험과 개선을 거쳤다는 것이었다. 그러나 그는 4.3BSD 배포판에 두 가지 구현 방식 모두를 포함시켜 커널에서 어떤 것을 사용할지 사용자가 직접 선택하도록 하는 결충안을 제시하였다.

マイ크 카렐스의 결정을 살펴본 뒤, DARPA는 두 가지의 코드 베이스를 릴리즈하는 것은 불필요한 상호 운영상의 문제를 발생시키며, 그렇기 때문에 하나의 구현만 릴리즈해야 한다고 결정하였다. 어느 코드 베이스를 사용할 것인지를 결정하기 위해 버클리와 BBN이 독립적인 제3자로 인정하는 BRL<sup>Ballistics Research Laboratory</sup>의 마이크 뮤스 Mike Muuse에게 두 코드를 모두 주었다. 한 달의 평가 후 BBN 코드가 시스템 부하 처리면에서는 버클리 코드보다 우수하다는 점을 제외하고는 전반적으로 버클리 코드가 효율적이라는 내용의 보고서가 나왔다. 결정적인 차이는 버클리 코드는 모든 시험을 무사히 통과한 데 비해 BBN 코드는 일부 열악한 조건에서 패닉이 일어난다는 것에 있었다. DARPA의 최종 결정은 4.3BSD가 버클리 코드 베이스를 유지한다는 것이었다.

마무리된 4.3BSD 시스템은 1986년 6월에 최종적으로 릴리즈되었다. 기대했던 대로 4.3BSD는 4.1BSD가 4BSD에 대해 그려했듯이 성능에 대한 많은 불만을 가라앉혀 주었다. 비록 대부분의 벤더는 시스템 V로 돌아서기 시작했지만 4.3BSD의 대부분은, 특히 네트워킹 서브 시스템은 시스템 V로 옮겨갔다.

1986년 10월, 키스 보스티 Keith Bostic이 CSRG에 합류하였다. 그의 고용 조건 하나는 이전에 작업하던 프로젝트를 끝낼 수 있도록 한다는 것이었는데, 바로 4.3BSD를 PDP-11로 이식하는 작업이었다. 카렐스와 나는 VAX에서 컴파일하여 250K가 되는 시스템을 PDP-11의 64K 주소 공간에 넣는다는 것은 불가능하다고 생각했지만 보스티가 자신의 시도를 끝낼 수 있도록 하는 데 동의하였다. 놀랍게도 복잡한 오버레이의 집합과 PDP-11에서 찾아낸 보조 프로세서 상태를 이용한 이식은

성공적이었다. 이 결과가 케이시 리덤Casey Leedom과 보스턴이 해낸 2.11BSD 릴리즈며, 1998년에도 여전히 사용된 최후의 PDP-11 일부에서 사용했다.

얼마 후 VAX 아키텍처는 수명을 다하고 있음이 점점 명백해졌으며 BSD를 실행하기 위한 다른 기계를 생각해야 했다. 그 당시 장래성 있는 새로운 아키텍처는 CCIComputer Consoles, Inc.에서 만든 것이었으며 Power 6/32라 불렸다. 불행히도 이 아키텍처는 회사가 전략적 방향을 바꾸기로 결정함에 따라 없어지게 되었다. 그러나 CSRG에는 몇 개의 기계를 제공하여 빌 조이가 시작하였던 BSD 커널을 기계의 존적인 부분과 비의존적인 부분으로 분리하는 작업을 끝낼 수 있었다. 이 작업의 결과는 1988년 6월 4.3BSD-Tahoe로 릴리즈되었다. Tahoe라는 이름은 CCI에서 Power 6/32라 릴리즈된 기계에서 사용되던 개발 이름에서 온 것이다. 비록 Power 6/32 기계 지원이 사용된 기간은 짧았지만 커널을 기계에 의존하지 않는 부분과 의존하는 부분으로 분리한 작업은 BSD가 여러 가지 다른 아키텍처로 이식되는 데 매우 큰 가치를 지녔던 일로 평가된다.

## 네트워킹, 릴리즈 1

4.3BSD-Tahoe가 릴리즈될 때까지 BSD를 받는 모든 사람은 AT&T 소스 라이선스를 먼저 얻어야 했다. 이는 베클리가 바이너리만의 형태로 BSD 시스템을 릴리즈 한 적이 없었기 때문이었다. 배포판은 항상 시스템 모든 부분의 완전한 소스 코드를 담았다. 특히 유닉스와 BSD 시스템의 역사는 사용자들이 소스 코드를 이용하는 것의 힘을 보여주었다. 사용자들은 수동적으로 시스템을 사용하지 않고 활발하게 버그를 수정하고 성능과 기능을 향상시켰으며, 완전히 새로운 기능을 추가하기도 하였다.

AT&T 소스 라이선스의 가격이 점점 높아짐에 따라 BSD 코드를 사용하여 PC 시장의 독립 TCP/IP 기반 네트워킹 제품을 제작하려던 벤더들은 바이너리당 비용이 엄청나다는 점을 알게 되었다. 따라서 그들은 베클리가 네트워킹 코드와 유틸리티

를 분리하여 AT&T 소스 라이선스를 요구하지 않는 라이선스 조항 아래에서 BSD를 제공하도록 요청하였다. TCP/IP 네트워킹 코드는 분명히 32/V에서는 존재하지 않았으며, 따라서 전체적으로 베클리와 그 공헌자들에 의해 개발된 것이었다. BSD에서 나온 네트워킹 코드와 지원 유ти리티는 네트워킹 릴리즈 1로 1989년 6월에 릴리즈되었으며, 이는 베클리 최초로 자유롭게 재배포할 수 있는 코드였다.

라이선스 조항은 자유로웠다. 라이선스를 받는 사람은 베클리와의 거래나 로열티 없이 소스 코드나 바이너리 형태로 수정되거나 수정되지 않은 형태로 릴리즈할 수 있었다. 유일한 요구 사항은 소스 파일의 저작권 문구를 유지하여야 하며, 이 코드를 통합한 제품은 문서 안에 해당 제품이 베클리 대학교와 공헌자들의 코드를 포함하고 있다고 알리는 것이었다. 베클리는 테이프를 얻는 데 1,000불의 요금을 받았지만, 누구나 이미 그것을 받은 사람에게서 무료로 복사본을 얻을 수 있었다. 사실 몇몇 큰 사이트는 릴리즈된 후 얼마 되지 않아 파일을 익명 FTP에 올려놓았었다. 쉽게 구할 수 있었지만 CSRG는 수백 개 기관에서 복사본을 구입한 금액이 이후 개발을 위한 자금이 되었다는 사실에 즐거워했다.

## 4.3BSD – Reno

얼마 후 기본 시스템의 개발이 재개되었다. 4.2BSD 아키텍처 문서에 처음 설명된 가상 메모리 시스템 인터페이스가 드디어 결실을 맺게 되었다. CSRG는 종종 그랬던 것처럼 처음부터 작성하기보다는 기존의 코드를 찾아 통합하려 했다. 그래서 새 가상 메모리 시스템을 설계하는 대신 기존의 대안을 찾아 나섰다. 첫 번째로 선택한 것은 썬Sun의 SunOS 가상 메모리 시스템이었다. 코드를 베클리에 기증하는 것에 대해 썬과 논의가 있었지만 그 논의는 소득이 없었다. 그래서 두 번째 선택안을 찾아보았는데 이것은 카네기 멜론 대학에서 만든 MACH 운영체제의 가상 메모리 시스템을 통합하는 것이었다. 유타 대학의 마이크 하이블러Mike Hibler는 MACH의 핵심 기술을 4.2BSD 아키텍처 매뉴얼에 설명된 사용자 인터페이스에 맞게 통합하

였다(이는 또한 SunOS에서 사용한 인터페이스였다).

당시 시스템에 추가된 또 다른 중요한 사항은 썬과 호환되는 버전의 네트워크 파일 시스템 NFS이었다. 또다시 CSRG는 실제 NFS 코드를 작성하지 않고 캐나다 구엘프 Guelph 대학의 릭 맥클렘 Rick MacLenn이 구현한 것을 얻었다.

4.4BSD를 발표할 만한 완전한 기능을 갖추지 않았지만 CSRG는 시스템에 새로 추가된 두 가지 중요 사항에 대한 부가적인 피드백과 경험을 얻기 위해 중간 릴리즈를 내기로 결정하였다. 라이선스를 갖는 중간 릴리즈는 4.3BSD-Reno라 불리었고 1990년 초에 발표되었다. 이 릴리즈는 네바다의 도박으로 유명한 도시의 이름을 따서 지어졌는데, 중간 릴리즈를 사용하는 것은 일종의 도박이라는 간접적인 암시였다.

## 네트워킹, 릴리즈 2

어느 날 CSRG의 주간 그룹 미팅 중 키스 보스틱은 자유롭게 배포할 수 있는 네트워킹 릴리즈의 인기에 대한 의제를 제시하였고, 더 많은 BSD 코드를 포함하는 확대된 릴리즈를 내는 것에 대한 가능성에 대해 문의하였다. 마이크 카렐스와 나는 보스틱에게 시스템의 많은 부분을 릴리즈하는 것은 너무 광범위한 작업이라는 점을 지적하였지만, 수백 개의 유ти리티와 C 라이브러리의 재구현을 어떻게 다룰 것 인지를 정리할 수 있다면 커널 수정 작업에 뛰어들 것이라는 데 동의하였다. 개인적으로 카렐스와 나는 그것이 이야기의 끝인 줄 알았다.

거기에서 멈추지 않고 보스틱은 대중의 네트워크에 기반한 개발 성과를 내는 기술을 개척하였다. 그는 사람들에게 알려진 설명에만 기반하여 처음부터 유닉스 유ти리티를 다시 작성해 달라고 요청하였다. 보상은 오직 다시 작성된 유ти리티의 이름 뒤의 버클리 공헌자 속에 자신의 이름이 들어가는 것이었다. 공헌은 천천히 시작되었으며 대부분 사소한 유ти리티에 대한 것이었다. 그러나 완성된 유ти리티의 수가

늘어가고 보스틱이 Usenix와 같은 공공 행사에서 프로젝트 기여에 대해 설명하자 기여된 유ти리티의 수는 늘어갔다. 곧 목록은 100여 개의 유ти리티를 넘었으며, 18개월 안에 거의 모든 중요한 유ти리티와 라이브러리가 재작성되었다.

보스틱은 자랑스럽게 목록을 손에 들고 마이크 카렐스와 나의 사무실로 걸어 들어와서 우리가 커널을 어떻게 할 것인지 알고 싶어했다. 우리의 작업에 따라 카렐스, 보스틱과 나는 이후 몇 달 동안 전체 배포판을 파일별로 들여다보고 32/V 릴리즈에서 고안된 코드를 제거하는 데 시간을 보냈다. 대략 정리가 되자 여섯 개의 커널 파일이 여전히 사용되며 쉽게 다시 사용할 수 없는 것이라는 사실을 알게 되었다. 이 여섯 개의 파일을 다시 만들어서 완전한 시스템을 릴리즈할 것을 고려하는 동안, 대신 지금까지 한 것을 릴리즈하자고 결정하였다. 그러나 이 확대된 릴리즈에 대해 대학 고위층으로부터 허락을 얻어야 했다. 많은 내부 토론과 독점적 코드를 판별하는 방법에 대한 겸증이 이루어지고 나서 릴리즈를 계속해도 좋다는 허락을 얻었다.

처음 생각은 두 번째로 자유롭게 재배포할 수 있는 릴리즈에 대해 완전히 새로운 이름을 짓는 것이었다. 그러나 우리는 대학 변호사들이 작성하고 인증한 완전한 새로운 라이선스를 얻는 일을 불필요한 자원의 낭비와 시간 지연으로 생각하였다. 그래서 새 릴리즈는 네트워킹 릴리즈 2라 부르기로 했는데, 인정된 네트워킹 릴리즈 1 라이선스 동의서를 단순히 교정만 했기 때문이었다. 이에 따라 두 번째의 아주 크고 자유롭게 배포할 수 있는 릴리즈는 1991년 6월에 발송되기 시작하였다. 재배포 조항과 비용은 첫 번째 네트워킹 릴리즈와 동일하였다. 이전처럼 수많은 개인과 조직이 베클리에게서 배포판을 얻으려면 1,000달러의 요금을 지불해야 했다.

네트워킹 릴리즈 2 배포판과 완전히 동작하는 시스템 사이의 간격을 좁히는 데는 오랜 시간이 걸리지 않았다. 릴리즈 후 6개월 동안 빌 졸리츠는 빠졌던 여섯 파일을 대체할 수 있는 파일을 작성하였다. 그는 곧 386/BSD라 불리는 386기반

PC 아키텍처에서 완전히 컴파일되고 부팅되는 시스템을 릴리즈하였다. 졸리츠의 386/BSD 배포판은 거의 전반적으로 네트워크에서 이루어졌다. 그는 배포판을 익명 FTP에 올리고 다운로드를 원하는 사람은 무료로 받을 수 있도록 하였다. 몇 주 안 되어 그의 동료는 크게 늘어났다.

불행히도 졸리츠는 다른 일을 해야 했으므로 밀려드는 386/BSD의 버그 수정과 개선안을 따라잡기에 필요한 시간을 투자할 수 없었다. 그래서 386/BSD의 릴리즈 후 몇 달 뒤, 열성적인 386/BSD의 사용자 그룹이 NetBSD 그룹을 결성하여 시스템을 유지하고 이후 개선하는 데 도움을 줄 공동의 자원을 모으기 시작하였다. 그들의 릴리즈는 NetBSD 배포판이라고 알려져 있다. NetBSD 그룹은 가능한 한 많은 플랫폼을 지원한다는 점을 강조하였으며, CSRG의 연구 스타일로 개발을 계속하였다. 1998년까지 이 배포판은 네트워크에서만 이루어졌고 달리 이용할 수 있는 어떤 배포판 매체도 존재하지 않았다. 이 그룹은 주로 철저한 테크니컬 사용자들을 목표로 한다. NetBSD 프로젝트에 대한 더 많은 정보는 <http://www.NetBSD.org>에서 찾을 수 있다.

NetBSD 그룹이 결성된 몇 달 뒤에 리눅스가 해왔던 것처럼 PC 아키텍처만을 지원하며, 기술적으로 다소 뒤처진 사용자 그룹을 따라간다는 목적으로 FreeBSD 그룹이 결성되었다. 그들은 정교한 설치 스크립트를 만들고 자신의 시스템을 저가의 CD-ROM으로 판매하기 시작하였다. 설치가 쉽다는 점은 네트워크와 컴텍스 같은 주요 무역 박람회에서의 대량 선전과 결합하여 빠르고 큰 성장 곡선을 그리게 하였다. 현재 FreeBSD는 분명히 모든 릴리즈 2의 파생 시스템 중 가장 많이 설치된 시스템이다.

FreeBSD는 또한 리눅스 바이너리를 FreeBSD 플랫폼에서 실행할 수 있도록 하는 리눅스 애플레이션 모드를 추가하여 리눅스의 인기에 편승하였다. 이 기능은 FreeBSD 사용자가 FreeBSD 시스템의 튼튼함, 신뢰성, 성능과 함께 지속해서 늘어

가는 리눅스용 애플리케이션을 사용할 수 있도록 한다. 이 그룹은 최근에 FreeBSD 몰(<http://www.FreeBSDmall.com>)을 열었는데, 여기서는 상담 서비스, 관련 제품 판매, 서적, 뉴스레터 등 FreeBSD 사용자들의 많은 물품을 취급한다.

1990년 중반, Open BSD가 NetBSD 그룹에서 분리되었다. 이들의 기술적인 초점은 시스템의 보안성을 향상시키는 것이고, 마케팅 초점은 시스템을 사용하기 쉽도록 하고, 더 폭넓게 이용할 수 있게 하는 것이었다. 따라서 FreeBSD 배포판의 여러 가지 쉬운 설치 기법에 대한 아이디어를 CD-ROM을 제작하여 판매하기 시작하였다. Open BSD 프로젝트에 대한 정보는 <http://www.OpenBSD.org>에서 찾을 수 있다.

## 소송

네트워킹 릴리즈 2 테이프를 기반으로 설계된 시스템을 자유롭게 재배포하기 위한 그룹의 형성과 더불어 버클리 소프트웨어 디자인사 Berkeley Software Design, Inc., 약칭 BSDI가 이 코드를 상업적으로 지원하는 버전을 개발하고 배포하려고 설립되었다(BSDI에 대한 더 많은 정보는 <http://www.BSDI.com>에서 찾을 수 있다). 다른 그룹과 비슷하게 BSDI는 자신의 시스템을 소스 코드와 바이너리를 포함하여 1992년 1월에 995달러에 판매하기 시작하였다. 이들은 이것이 시스템 V 소스 코드에 바이너리 시스템을 더한 가격에 99%를 할인한 가격이라며 광고하기 시작했다. 그 당시 (미국)1-800-ITS-Unix에 전화하면 광고를 들어볼 수 있었다.

BSDI가 세일즈 캠페인을 시작한지 얼마 안 되어 유닉스 시스템 연구소(USL, 유닉스를 개발하고 판매하기 위해 분리된 AT&T의 자회사)에서 온 편지를 받았다. 이 편지는 그들의 제품을 유닉스로 선전하는 것을 중지할 것과, 특히 오해를 살 수 있는 전화번호의 사용을 중지할 것을 요구하였다. 전화번호는 곧 사용이 중지되었고 광고는 이 제품이 유닉스가 아니라는 설명을 포함하도록 변경되었지만, USL은 여전히 만족하지 못했고 BSDI가 제품을 판매하지 못하게 하는 소송을 제기했다.

USL은 소장에서 BSDI 제품이 USL의 독점 소스 코드와 영업 기밀을 포함한다고 주장하였다. USL은 소송이 종결될 때까지 BSDI에 대한 법원의 판매 금지 명령을 얻어내려고 시도했고, BSDI 배포판이 계속 판매된다면 영업 기밀이 누출되어 회복할 수 없는 피해를 당할 것이라고 주장하였다.

판매 금지에 대한 예심에서 BSDI는 단순히 버클리 대학이 자유롭게 배포하는 소스 코드에 여섯 개의 추가적인 파일을 사용할 뿐이라고 주장했다. 그들은 추가된 여섯 개 파일의 내용에 대해서는 기꺼이 토론하고 싶어했지만, 버클리 대학이 배포하는 파일에 대해서도 책임져야 한다고는 생각하지 않았다. 판사는 BSDI의 주장에 동의하여 USL 측에 고소장을 여섯 개 파일에 대해서만 다시 작성해야 하고, 그렇지 않으면 고소를 기각할 것이라는 점을 알렸다. 단지 여섯 개 파일에 대한 소송으로는 승소하기가 어렵다는 것을 알게된 USL은 BSDI와 버클리 대학 모두에 대한 소송을 다시 제기하기로 결정하였다. 그리고 이전처럼 USL은 버클리 대학의 네트워킹 릴리즈 2와 BSDI 제품의 판매 금지를 요청하였다. 몇 주 후에 다가올 법원의 판매 금지 명령에 대한 심리를 앞두고 그에 대한 본격적인 준비가 시작되었다. 거의 모든 사람이 BSDI에 고용되었기 때문에 CSRG의 모든 회원이 증언대에 섰다. 변호사 사이에서는 개요, 반증, 반증의 반증이 오갔다. 키스 보스티과 나는 개인적으로 여러 가지 반증에 도움이 될 수백 페이지의 자료를 작성해야 했다.

1992년 12월, 미합중국 뉴저지 지방 법원의 디킨슨 데베브와즈 Dickinson R. DeBevoise 판사는 판매 금지 명령에 대한 의견 심리를 하였다. 판사는 보통 판매 금지 명령 요청에 대해서는 즉시 판결을 내리지만 이 건에 대해서는 좀 더 숙고하기로 결정했다. 6주 후 금요일, 판사는 판매 금지 명령에 대한 요청을 기각하고, 두 가지 사항을 제외한 모든 고소 내용을 인정하지 않았다. 고소 내용 중 인정된 두 가지 사항은 최근의 저작권과 영업 기밀 누출 가능성에 대한 것이었다. 또한 판사는 연방 법정에서 사건을 심리하기 전에 주 법정에서 심리할 것을 제안하였다.

이 판결로 입장이 유리해진 베클리 대학은 다음 월요일 아침 USL에 대해 맞대항하는 고소장을 캘리포니아 주 법원에 제출하였다. 캘리포니아에서 먼저 제기됨에 따라 베클리는 이후 주 법정 소송의 장소를 확보하게 되었다. 미 헌법은 부유한 피고가 모든 주마다 고소함으로써 상대를 금전적으로 곤란에 빠지게 하는 것을 막기 위해 한 사건에 대한 소송이 하나의 주에서만 이루어지도록 요구한다.

베클리 대학은 고소장에서 USL이 베클리 대학과 맷은 라이선스에서 요구하는 것처럼 시스템 V 안 BSD 코드의 사용에 대해 대학의 정해진 크레딧 문구를 표시해야 하는 의무를 지키지 못했다는 점을 주장하였다. 이 주장이 유효하다는 판결을 받게 되면 베클리 대학은 USL에게 적절한 크레딧을 추가하여 USL의 모든 문서를 다시 인쇄하도록 요구할 수 있으며, 그들의 라이선스를 받는 모든 사람에게 그 잘못을 알리고 월 스트리트 저널이나 포천지와 같은 주요 출판물에 부주의한 잘못에 대한 시정 광고를 내도록 강제할 수 있었다.

주 법원에 소송을 제기한 뒤, 곧 USL은 AT&T에서 노벨로 매각되었다. 노벨의 CEO인 레이 누어다 Ray Noorda는 법정이 아닌 시장에서 경쟁하겠다고 공식적으로 밝혔다. 1993년 여름, 화해의 대화가 시작되었다. 불행히도 법정 공방으로 인한 양측의 골이 너무 깊었기 때문에 대화는 느리게 진행되었다. USL 측의 레이 누어다의 도움으로 많은 장애 문제가 해결되었고 1994년 1월, 마침내 화해를 하게 되었다. 결과는 네트워킹 릴리즈 2를 구성하는 18,000개의 파일에서 3개의 파일을 지워야 하며, 다른 파일에도 몇 가지 사소한 변경을 해야 한다는 것이었다. 또한 베클리는 약 70여 개의 파일에 USL 저작권을 추가하는 데는 동의하였지만, 이 파일은 계속 자유롭게 재배포할 수 있었다.

## 4.4BSD

새로운 릴리즈는 4.4BSD-Lite라 불리며 1994년 6월 네트워킹 릴리즈에 사용되었던 것과 동일한 조항 아래 릴리즈되었다. 특히 소스 코드와 바이너리 형태의 자

유로운 재배포를 허용하는 조항에는 저작권을 그대로 버클리가 가지며, 다른 사람들이 코드를 사용할 경우에는 버클리와 그 공현자를 포기해야 한다는 제한이 있었다. 동시에 완전한 소스 코드가 4.4BSD-Encumbered로 릴리즈되었지만 여전히 USL 소스 라이선스를 얻어야만 소스를 받을 수가 있었다.

소송 중재안에는 USL이 4.4BSD-Lite를 시스템의 기반으로 사용하는 어떤 조직에 대해서도 소송을 제기할 수 없다는 점도 규정되어 있었다. 따라서 당시 릴리즈를 내던 모든 BSD 그룹인 BSDI, NetBSD, FreeBSD는 자신의 코드 기반을 4.4BSD-Lite로 수정하고 그들 고유의 개선점과 향상점을 다시 통합해야 했다. 이 재통합으로 여러 BSD 시스템의 개발이 단기적으로 지연되었지만 분기했던 모든 그룹이 네트워킹 릴리즈 2 이후 CSRG에서 이루어졌던 3년간의 개발 성과를 다시 동기화도록 하였다는 점에서 외면상 불행해 보이는 행복이었다.

## 4.4BSD – Lite 릴리즈 2

4.4BSD-Encumbered와 4.4BSD-Lite 릴리즈에서 얻은 수익은 버그 수정과 향상점을 통합하는 파트타임 작업의 기금으로 사용되었다. 이런 변화는 버그 리포트와 기능 향상의 빈도수가 아주 낮아질 때까지 2년간 계속되었다. 최종 변경 사항은 1995년 6월에 4.4BSD-Lite의 릴리즈 2로 릴리즈되었다. 대부분의 변경 사항은 최종적으로 다른 시스템의 소스 코드 기반에도 포함되었다.

4.4BSD-Lite 릴리즈 2가 릴리즈된 후에 CSRG는 해체되었다. 거의 20년간 BSD를 이끌어왔던 우리는 이제 신선한 아이디어와 끝없는 열정을 가진 새로운 사람들에게 이 일을 맡길 때라고 느꼈다. 시스템 개발을 전체적으로 살필 수 있는 하나의 중심적인 권위가 가장 좋기는 했지만, 여러 목표를 가진 여러 그룹의 아이디어는 서로 다른 많은 접근 방법이 시도될 수 있도록 보장해준다. 시스템은 소스 코드 형태로 릴리즈되므로 가장 좋은 아이디어를 다른 그룹이 쉽게 채택할 수 있다. 그 중 한 그룹이 특별히 유력해진다면 그들이 결국 지배적인 시스템이 될 것이다.

오늘날 오픈 소스 소프트웨어 운동에 대한 관심과 주목이 날로 늘고 있다. 리눅스 시스템이 가장 유명하기는 하지만 패키징되는 절반 정도의 유틸리티는 BSD 배포판에서 나온 것이다. 리눅스 배포판은 또한 컴파일러, 디버거, 다른 개발 도구를 자유 소프트웨어 재단에서 작성한 것에 크게 의존한다. CSRG, 자유 소프트웨어 재단, 리눅스 커널 개발자는 모두 오픈 소스 소프트웨어 운동이 시작되었던 플랫폼을 마련해 주었다. 나는 오픈 소스 소프트웨어 운동을 개척하는 데 도움을 줄 수 있는 기회를 얻었던 것을 매우 자랑스럽게 생각하며, 오픈 소스 소프트웨어가 모든 곳의 사용자와 회사가 소프트웨어를 개발하고 구입하기 위해 선호하는 방법이 되는 날을 기대한다.

### 3 | 인터넷 엔지니어링 태스크포스

스콧 브래드너 Scott Bradner

송창훈 역

IETF<sup>Internet Engineering Task Force</sup><sup>01</sup>는 존재하지 않는 것에 큰 영향을 미쳐왔다. TCP/IP 자체는 논외로 하더라도 인터넷의 기반이 되는 모든 기술은 IETF 안에서 개발되고 개선돼왔기 때문이다. IETF 워킹그룹은 라우팅, 관리, 전송 표준을 만들었는데, 이러한 표준이 없었다면 인터넷은 존재하지 못했을 것이다. 또한 IETF 워킹그룹은 인터넷의 안전에 기여할 보안 표준과 인터넷을 좀 더 예측 가능한 환경으로 만들기 위한 서비스 품질에 관한 표준, 그리고 차세대 인터넷 프로토콜 그 자체를 위한 표준을 정의해왔다.

IETF 표준은 매우 성공적이었다. 인터넷은 단일 기술로는 역사상 그 어떤 기술보다 빠르게 성장하고 있으며 철도나 전등, 전화, 텔레비전의 발전 속도를 훨씬 능가하고 있다. 그러나 이는 단지 시작일 뿐이다. 이 모든 것은 자발적으로 정의된 표준과 함께 이루어졌으며, 표준의 수용 또한 자발적인 것이기 때문에 어떠한 나라도 IETF 표준의 사용을 요구받지 않는다. 세계 여러 정부가 강행한 것들을 포함해서 경쟁하는 표준이 새롭게 나타나고 사라지는 과정을 통해 IETF 표준은 더욱 확고하게 성장했다. 물론 IETF 표준이 모두 성공적이었던 것은 아니다. 현실 세계의 구체적 요구에 부합하며 쓸모 있는 것만이 살아남아서 이름 그대로 진정한 표준이 되었다.

IETF와 그 표준이 성공적일 수 있었던 이유는 오픈 소스 공동체가 성공하고 있는 것과 같은 맥락에서 찾을 수 있다. IETF의 표준화 과정은 관심 있는 사람 누구나

참여할 수 있는 개방적인 형태로 이루어지며 모든 IETF 문서는 인터넷을 통해 자유롭게 이용하거나 복제할 수 있다. 사실 IETF의 공개적인 문서화 과정은 오픈 소스 운동의 가능성을 엿볼 수 있는 사례라고 할 수 있다.

이 글은 IETF에 대한 간략한 역사를 소개하고 IETF의 구성과 업무 처리 방법을 검토한 뒤에 마지막으로 공개 표준과 공개 문서 그리고 오픈 소스에 대한 몇 가지 중요한 생각을 덧붙여 보려고 한다.

## IETF의 역사

IETF는 미국 정부로부터 기금을 지원받던 연구자들의 분기별 회의로 1986년 1월부터 시작되었다. 같은 해 10월에 있었던 4차 회의에 비정부 업체 대표들이 초청되었고, 이때부터 모든 IETF 회의는 참석을 희망하는 사람 모두에게 공개되었다. 초기 단계에는 규모가 크지 않아 처음 5번의 회의에 평균적으로 35명 미만이 참석했고 13번의 회의 중 참석자가 가장 많았던 1989년 12차 회의의 경우에도 120명이 왔을 뿐이다. 하지만 IETF는 그 후 크게 성장하여 1992년 3월 23차 회의에 500명이 넘게 참석했고 1994년 3월 29차 회의에는 750명이 넘게, 1994년 12월 31차 회의에는 1,000명이 넘게 참석하면서 1996년 12월 37차 회의에는 참석자가 거의 2,000명에 육박하게 되었다. 이때부터 증가율이 주춤해져 1998년 12월 43차 회의에 2,100명이 참석했는데, 이러한 과정에서 IETF는 1991년부터 정례 회의 횟수를 연간 4회에서 3회로 줄이게 되었다.<sup>02</sup>

IETF는 미국 버지니아주 레스턴 Reston 외곽에 작은 사무국<sup>03</sup>을 두고 있고 RFC 편집자<sup>04</sup> 업무는 서던캘리포니아대학교 University of Southern California 정보과학연구소 Information Sciences Institute, ISI에서 수행한다.

IETF는 지금까지 단 한 번도 법인 형태로 구성되지 않았다. 법적인 실체 없이 단지 활동만 해왔을 뿐이다. IETF 운영비용은 1997년 말까지 미국 연방정부 기금과

IETF 회의 참가비<sup>05</sup>로 충당했으며, 1998년부터는 ISOC<sup>Internet SOCIETY<sup>06</sup></sup>의 지원과 IETF 회의 참가비로 충당하고 있다.

1992년에 만들어진 ISOC의 목적 중 일부는 표준화 활동이 원만하게 진행될 수 있도록 IETF에 법적 보호와 사업 기금을 제공하는 것이다. ISOC는 회원제로 운영되는 국제적인 비영리 기구이며 인터넷이 보급되지 않은 국가에 인터넷을 전파하기 위한 활동도 한다.<sup>07</sup> 이제 IETF를 가장 정확하게 정의한다면, ISOC의 지원 아래 표준화를 진행하는 기구라고 말할 수 있다.

## IETF의 구성과 특성

IETF는 회원에 의해 움직이지만, 회원의 자격과 구분을 정하지 않은 기구라고 설명할 수 있다. 즉 회원 제도가 없기 때문에 회원사나 회원 단체는 성립하지 않으며, 개인 자격으로 IETF 회의나 메일링리스트에 참여하는 사람 모두가 IETF 회원이다.<sup>08</sup>

이 글을 쓰는 1998년 현재 IETF에는 공식적으로 115개 워킹그룹이 존재하며<sup>09</sup>, 워킹그룹은 애플리케이션Applications, 일반General, 인터넷Internet, 운영 및 관리Operations and Management, 라우팅Routing, 보안Security, 전송Transport, 사용자 서비스User Service의 8개 분야<sup>10</sup>로 편성되어 있다. 각각의 분야는 1명 또는 2명의 자원자가 분야 책임자Area Director가 되어 운영을 맡고 있으며 분야 책임자들은 IETF 의장과 함께 운영 위원회Internet Engineering Steering Group, IESG를 구성한다. IESG는 IETF가 만든 표준을 승인하는 역할을 한다. 또한 12명의 인원으로 구성된 인터넷 아키텍처 위원회Internet Architecture Board, IAB는 워킹그룹의 조직 형태와 작업 결과에 대한 자문을 IESG에 제공한다.

분야 책임자와 IAB 위원의 임기는 2년이며 선거인단에 의해 선출된다.<sup>11</sup> 선거인단은 분야 책임자와 IAB 위원을 선출할 시점을 기준으로 이전에 열렸던 3번의 IETF 회의에서 최소한 2번 이상 참석한 사람 중에서 자원자를 무작위로 추첨하여 매년 새롭게 구성한다.

## IETF 워킹 그룹

IETF와 다른 표준화 기구들 사이의 주된 차이 중 하나는 IETF가 매우 상향적인 구성 방법을 갖고 있다는 점이다. IESG나 IAB가 중요하다고 판단한 현안을 처리하기 위해 워킹그룹을 직접 만드는 일은 거의 없다. 대부분의 워킹그룹은 특정한 문제에 관심 있는 사람들이 소규모 모임을 먼저 만든 후에, 분야 책임자에게 워킹그룹 생성을 제안하는 과정을 통해 이루어진다. 이는 IETF가 미래 과업 계획을 주도적으로 만들 수 없다는 것을 의미하지만 다른 한편으로는 워킹그룹을 성공적으로 이끌기 위한 회원들의 열정과 전문 지식을 충분히 보장해 주는 방법이라고 할 수 있다.

새로운 워킹그룹을 만들기 위해서는 워킹그룹에 대한 현장Charter<sup>12</sup>을 작성해야 하는데, 이 작업은 워킹그룹 생성 제안자들과 해당 분야 책임자 사이의 협의로 이루어진다. 현장에는 워킹그룹에 필요한 조건, 다른 그룹과의 교류 방법, 그리고 활동 목적과 일정 및 범위가 포함되어야 한다. 현장 작성이 완료되면 IESG와 IAB 메일링리스트로 올려져 검토가 진행된다. 일주일 동안 심각한 이의가 제기되지 않으면, 유사한 종류의 작업이 진행되는지를 확인하기 위해 IETF 전체 메일링리스트와 다른 표준화 기구와의 의사 교환을 위한 메일링리스트에 올려진다. 그 후 다시 일주 일간 추가적인 의견 수렴을 거친 뒤에 IESG가 현장을 승인하면 새로운 워킹그룹이 만들어진다.

## IETF 문서

모든 IETF 문서는 자유롭게 이용할 수 있도록 인터넷으로 공개한다. 이를 보장하기 위해서 IETF는 (저자가 미래에 문서를 철회할 수 없도록) 문서를 발간할 때 저자로부터 제한된 저작권을 양도받는다. 이 제한된 저작권 아래에서는 누구든지 문서 전부를 재발간할 수 있으며, IETF는 표준화 과정에서 대부분의 문서에 대한 2차적 문서를 만들 수 있다. 그 외의 다른 권리는 모두 저자가 가진다.

IETF 표준화 활동 중에 RFC 편집자에 의해 (파일 형태로) 인터넷에 정식 발간되는 문서를 통칭 RFC라 한다. RFC는 원래 특정 사안에 대해 다른 사람의 견해를 묻는다는 뜻인 ‘Request For Comments’의 약자이기 때문에 밀도 높은 검토 과정을 거쳐 발간된 문서에 대해서도 RFC, 즉 ‘의견 요청’이라 부르는 데는 무리가 있다. 따라서 RFC는 마치 고유명사처럼 RFC라는 이름 그 자체로 받아들이는 것이 바람직하다.

IETF의 표준화 과정은 문서를 만들고 완성해 가는 과정이라 할 수 있다. 제안을 담은 문서가 워킹그룹에 의해 작성된 후 회람되고 협의되며 다듬어지는 과정을 통해 승인되면 정식 RFC로 발간된다. 그 후 같은 노력이 다음 단계의 표준화 과정(RFC 발간 과정)으로 계속 이어지고, 이것은 RFC, 즉 RFC가 담고 있는 내용이 숙성되어 완전한 정식 표준으로 승인될 때까지 반복된다. RFC는 정식 표준화 과정인 표준 트랙 Standards Track과 비표준 트랙 Non-standards Track 두 가지 범주의 문서로 크게 구분된다. 표준 트랙 안에서의 제안은 표준화 진행 단계에 따라 ‘제안 표준 Proposed Standard’, ‘준 표준 Draft Standard’, ‘정식 표준 Internet Standard’의 지위를 가지며 같은 이름의 RFC로 출판된다. 비표준 트랙 안에서의 문서 지위는 ‘실험 Experimental’, ‘참고 Informational’, ‘역사 Historic’가 있다.

그 외의 문서로 ‘인터넷 초안 Internet-Drafts’이 있다. 이 문서는 RFC의 원래 의미인 ‘Request For Comments’의 목적으로 사용되는 임시 문서다. ‘인터넷 초안’은 언제든지 없어질 수 있기 때문에 작업을 진행하는 중이 아닌 한 인용하거나 참조하지 않으며 6개월 뒤에 자동으로 삭제된다.

## IETF 표준화 과정

IETF의 모토는 ‘대략적인 합의와 작동하는 코드 rough consensus and running code’<sup>13</sup>다. 특정 안건이 채택되는 데 워킹그룹의 만장일치 찬성이 요구되지는 않지만 대부분의 구성원에게 그 필요성을 입증하지 못하면 승인될 수 없다. 안건이 승인되는 데

필요한 정해진 찬성률은 없다. 그러나 일반적으로 90% 넘는 찬성을 얻으면 승인되고 80% 미만일 경우에는 부결되는 모습을 흔히 볼 수 있다. 따라서 ‘대략적 합의’는 정확한 숫자는 아니지만 대단히 많은 다수가 찬성한다는 의미다. 대부분의 다른 표준화 기구에서는 투표나 만장일치에 가까운 합의에 의해서만 안건이 승인되지만 IETF는 정례 회의에서 실제 투표를 하지 않는다. 필요한 경우, 거수<sup>14</sup>를 통해 합의가 명확한지 가늠하기는 하지만 이것이 결정력 있는 투표를 의미하지는 않는다. 또한 완전한 표준으로 승인되기 이전에 실제로 작동하는 구현이 이루어져야 하는데, 이 두 가지가 ‘(안건에 대한) 대략적인 합의와 (제안된 표준에 대한 실제로) 작동하는 코드’의 명확한 의미다.

비표준 트랙의 문서는 자기의 생각이나 기술 제안을 인터넷 공동체에 알리고 싶은 개인이나 IETF 워킹그룹 활동을 통해 만들어진다. RFC로 발간될 거의 모든 제안은 IESG의 검토를 거치며, 그 후 IESG는 RFC 편집자에게 제안의 발간 타당성에 대한 자문을 제공한다. 이런 과정을 거쳐 RFC 편집자는 해당 제안을 RFC로 발간할지 결정하는데, IESG가 주석을 제공한 경우라면 RFC에 주석을 포함시킬지도 결정한다. IESG가 도움이 될 것으로 판단해 일종의 주의 사항을 주석으로 제공했다면, 이는 해당 제안이 불만족스러웠다는 것을 의미한다.

표준 트랙 문서의 일반적인 예는 IETF 워킹그룹이 만든 ‘인터넷 초안’이 RFC로 발간되는 것이다. 제안을 마지막으로 검토하는 과정을 라스트 콜<sup>Last call</sup>이라고 하는데, 라스트 콜은 워킹그룹 의장의 주관으로 제안에 문제될 만한 사항이 있는지 확인하는 과정이다. 메일링리스트를 통해 보통 2주간의 일정으로 진행한다. 만약 라스트 콜을 통해 워킹그룹 대다수가 찬성하는 대략적인 합의로 승인되면 제안은 IESG로 보내진다.

IESG 검토의 첫 단계는 라스트 콜을 IETF 전체로 확대하는 것이다. IESG 라스트 콜은 IETF 전체 공지 메일링리스트로 보내져 해당 내용을 놓친 사람도 다시 의견

을 개진할 수 있도록 한다. 일반적으로 IETF 워킹그룹이 만든 제안의 라스트 콜에는 2주의 시간이 소요되며, 그렇지 않은 경우에는 4주의 시간이 소요된다.

IESG는 IETF 전체 라스트 콜의 결과를 제안에 대한 심의 기준으로 삼는다. 그 후 제안을 승인하여 RFC 편집자에게 발간을 요청할 수도 있고, 심의 결과를 토대로 문서를 개정하기 위해 원저자에게 돌려보낼 수도 있는데, 표준 트랙 안의 모든 단계에는 동일한 기준을 적용한다. 즉, ‘제안 표준’에서 ‘준 표준’을 거쳐 ‘정식 표준’으로 성숙하는 과정에 모두 동일한 기준을 적용한다.

제안들은 일반적으로 표준 트랙에서 ‘제안 표준’의 지위로 논의를 시작하지만, 기술에 불확실한 점이 있거나 추가적인 확인이 필요하다고 판단될 때는 비표준 트랙에서 ‘실험’ 지위로 만들어 논의할 수 있다. 따라서 표준 트랙의 ‘제안 표준’은 알려진 기술 결합이 없는 좋은 아이디어라는 의미다. 최소 6개월간의 ‘제안 표준’ 지위에서 논의를 거친 제안은 ‘준 표준’ 지위를 가진 것으로 간주된다. ‘준 표준’ 지위의 제안은 반드시 문서의 내용이 명확하고 제안과 관련된 어떠한 지식재산권 문제도 양해되었거나 해결 가능함을 입증해야 하는데, 이는 지식재산권이 존재한다면 별도의 이용허락 실행 절차를 거쳐 최소한 두 개 이상의 서로 다른 코드로 개발한 독립적이고 상호운용이 가능한 실제적인 구현을 요구하는 것으로 이루어진다. 여기서 유의할 점은 프로토콜의 모든 독립 기능은 다중 구현을 요구받는다는 것이다. 이러한 요구를 충족하지 못한 기능은 제외된다. 따라서 IETF 표준은 표준화 단계가 진행될수록 더 간소해진다. IETF 모토의 뒷부분인 ‘작동하는 코드’란 바로 문서상의 제안으로 표준화가 진행되지 않고 이렇게 실제로 구현되는 것을 말한다.

IETF 표준화 과정의 마지막 단계는 ‘정식 표준’이다. 최소한 4개월 이상의 ‘준 표준’ 지위에서 논의를 거치며 충분한 시장성을 입증하면 제안은 마침내 완전한 ‘정식 표준’으로 간주된다.

다른 표준화 기구의 표준화 진행 과정과 비교할 때 IETF에는 두 가지 두드러진 차이점이 있다. 첫째는 최종적으로 결정된 표준의 핵심 대부분이 ‘제안 표준’ 상태와 거의 동일하다는 점이다. 왜냐하면 ‘제안 표준’은 실제 구현이 되지 않았을 뿐 ‘인터넷 초안’ 상태부터 많은 검증을 거쳐 이미 문제가 없다고 판단된 유용한 기술이기 때문이다. 둘째는 만장일치 대신 사용되는 대략적인 합의다. 만장일치에서는 찬반 논쟁으로 소란을 일으키는 사람을 배제하려고, 기능을 덜 추가하는 제안이 만들 어질 수 있다.<sup>15</sup>

간단히 정리하면, IETF는 상향식 과업 생성 방식으로 운영되며, ‘선 비행, 후 구매 fly before you buy’<sup>16</sup> 원칙을 믿는다.

## 공개 표준, 공개 문서 그리고 오픈 소스

IETF 표준이 지금까지 성공적이었던 주된 요인 중 하나는 IETF 문서와 표준화 과정에 대한 공개 정책이라고 할 수 있다. IETF는 메일링리스트와 회의 뿐 아니라 문서 자료를 모두 공개하는 매우 드문 주요 표준화 기구의 하나다. 기존의 많은 표준화 기구는 물론이고 심지어 인터넷 관련 신생 단체 중 일부도 회의와 문서 자료를 회원에게만 제공하거나 요금을 지불해야만 이용할 수 있게 한다. 때로는 표준화 기구 운영 기금 마련을 위한 목적으로 이런 방법이 사용되기도 하며, 기구 자체가 회비 납부를 기본으로 하는 회원제로 운영되기도 한다. 사람들이 이러한 기구에 가입 하려는 이유 중 하나는 표준화 과정에 참여하거나 이러한 기구가 진행하는 표준에 접근하기를 원하기 때문이다. 표준화 과정에 대한 참여를 제한하면 그렇지 않은 경우에 비해 사용자와 기업의 필요를 충족하지 못하는 표준을 만들 가능성이 커지고, 운영자가 표준을 합리적으로 지원하지 못할 정도로 복잡해질 가능성 또한 커진다고 할 수 있다. 표준화를 진행하는 문서에 대한 접근을 제한하면 표준을 구현할 사람이 표준에 있는 특정 기능의 기원과 원리가 무엇인지 이해하기 더 어렵게 만들기 때문에 결과적으로 결함 있는 구현으로 이어질 수 있다. 또한 완성된 표준에 대한

접근을 제한하면 학생과 개발자가 표준을 이해하고 활용하려고 할 때 미미한 출발점에 설 수밖에 없게 된다.

IETF는 오픈 소스 운동이 형성되기 오래전부터 이미 오픈 소스의 개념을 지원해 왔다. 최근까지도 표준 트랙 과정에서 다중 구현의 일부로 IETF 기술에 대한 참조 구현<sup>17</sup>을 요구하는 것이 보통이었다. 이것이 IETF 표준화 과정에 공식적으로 포함된 적은 없었지만, 일반적으로 매우 유용한 부산물이었다. 불행하게도 표준은 점점 더 복합해지고 구현에 필요한 비용은 늘고 있어 참조 구현은 이제 쇠퇴하고 있다. 이러한 추세는 지속되겠지만, 오픈 소스 운동이 IETF 표준화 과정의 비공식 부분에 새로운 활력을 불어넣는다면 매우 좋을 것이다.

눈앞에 뚜렷이 보이지 않을지는 몰라도 공개 문서와 공개된 표준화 과정은 오픈 소스 운동에 있어 필요 불가결한 부분이다. 일반적으로 표준 문서 안에 명시한 현재 진행 중인 일에 대한 명확한 합의가 없다면, 오픈 소스 운동처럼 분산적으로 진행되는 프로젝트는 혼란을 겪거나 실패하기 쉬울 것이다. 공개 문서와 공개된 표준화 과정 그리고 오픈 소스 사이에는 고유한 협력 관계가 존재한다. 이것이 지금의 인터넷을 만들었고, 미래의 또 다른 경이로움을 만들 것이다.

- 
- 01 역자주\_‘태스크포스’란 특정 작업을 수행하기 위해 만든 팀을 말한다. ‘전담대응팀’으로 표현하기도 하지만 이 번역문에서는 태스크포스를 그대로 사용했다.
  - 02 역자주\_최근의 IETF 회의는 2013년 7월 87차 베를린 회의로 1,373명이 참석했다. 이는 1,432명이 참석한 2005년 7월 63차 파리 회의 이후 최대 참석자 수이며, 최근의 IETF 회의 참석자 수는 평균 1,200명 정도의 수준이다. 한국에서는 2004년 2월 59차 회의가 서울에서 열렸다. 참석자 수는 1,400명이었으며, 이 중 한국 참석자는 약 500명이다(제2차 북핵 위기를 외국인 참석 저조 원인으로 꼽기도 한다). 지난간 IETF 회의 정보는 <http://www.ietf.org/meeting/past.html>에서 참고할 수 있다.

- 03 역자주\_IETF 사무국(<http://www.ietf.org/secretariat.html>)은 2013년 현재 캘리포니아주 프리몬트(Fremont, California)에 소재하고 있으며, IETF 회의 준비와 같은 사무관리 지원이 주된 업무다.
- 04 역자주\_RFC는 IETF가 정식으로 발간하는 문서의 통칭이다. 이에 대해서는 본문 뒷부분에서 자세히 설명한다. RFC로 발간할 문서를 최종 점검하고 이를 인터넷으로 공개하는 일을 맡은 사람을 RFC 편집자(RFC Editor)라고 한다. 인터넷에서 RFC를 검색할 수 있도록 색인 파일을 관리하는 일도 한다. 1998년 10월 16일 향년 55세로 사망할 때까지 ISI의 조너선 포스텔(Jonathan B. Postel, 1943~1998) 박사가 RFC 편집자였으나, 그 후 민간회사로 위탁·이관되면서 업무 또한 여러 명이 함께 담당하고 있다. 이런 이유 때문에 2000년 종이책에서는 'RFC 편집국'으로 번역했으나, 2013년 전자책에는 역사적 이유를 고려해 RFC 편집자로 그대로 옮긴다. IANA와 ISOC 활동에 핵심적인 역할을 했던 포스텔 박사의 추모 페이지는 <http://www.postel.org/postel.html>이며 RFC 편집자 홈페이지는 <http://www.rfc-editor.org>다. 2013년 현재 RFC 사무국 업무와 RFC 편집자 업무는 같은 위탁업체가 맡고 있다.
- 05 역자주\_2013년 현재 IETF 회의 참가비는 할인 없는 일반 가격 기준으로 미화 800달러(약 89만 원)이다.
- 06 역자주\_Internet Society는 인터넷 협회 또는 인터넷 협의회쯤으로 번역할 수 있지만, 약어인 ISOC가 널리 통용되고 있으므로 별다른 번역 없이 ISOC라고 표기한다. 인터넷 소사이어티로 그대로 옮기기도 한다.
- 07 역자주\_ISOC 홈페이지(<http://www.internetsociety.org/internet/what-internet/facts-and-figures>)에서 다양한 세계 인터넷 통계를 확인할 수 있다.
- 08 역자주\_기업이나 단체도 개인 자격으로 참여할 수 있다.
- 09 역자주\_2013년 기준으로 125개 워킹그룹이 존재한다.
- 10 역자주\_사용자 서비스 분야는 없어졌고 2005년 12월 64차 벤쿠버 회의에서 '실시간 애플리케이션 및 인프라(Real-time Applications and Infrastructure)' 분야가 신설돼 모두 8개 분야가 존재한다. RAI 분야는 VoIP와 같은 IP 통신에 대응하기 위한 것이다. IETF 워킹그룹에 대한 정보는 <http://datatracker.ietf.org/wg/>에서 참고할 수 있다.
- 11 역자주\_선출 후 ISOC 이사회의 승인 절차가 뒤따른다.
- 12 역자주\_현장은 워킹그룹 자체의 운영 규범이면서 워킹그룹을 만들기 위한 승인서의 성격도 함께 가진다. 현장은 charter라는 단어로 표현하며 IESG로 제출된 charter(명사로 쓰임)를 검토해서 워킹그룹의 생성을 승인해주는 것을 charter(동사로 쓰임)한다고 표현한다. IETF에 125개의 워킹그룹이 있다면 125개의 charter된 워킹그룹이 있다고 표현할 수 있다. 현장과 워킹그룹에 대한 사항은 RFC 1603(<http://www.rfc-editor.org/info/rfc1603>)을 통해 자세히 참고할 수 있다.

- 13 역자주\_1992년 7월 24차 IETF 회의에서 데이비드 클라크(David D. Clark)가 발표한 내용 중 일부로 IETF의 문화를 상징하는 표현으로 간주된다. 이 표현이 들어간 전체 문장은 “우리는 왕과 대통령, 투표를 거부한다. 우리는 대략적인 합의와 작동하는 코드를 믿는다(We reject kings, presidents and voting. We believe in rough consensus and running code)”이다. [http://groups.csail.mit.edu/ana/People/DDC/future\\_ietf\\_92.pdf](http://groups.csail.mit.edu/ana/People/DDC/future_ietf_92.pdf)에서 발표 자료 전문을 참고할 수 있다.
- 14 역자주\_거수 이외에 IETF 회의에서 대략적인 합의를 살피기 위해 사용하는 또 하나의 방법으로 허밍(hum)이 있다. 안건에 대해 찬성하는 사람은 ‘허밍’이라는 의장의 말에 따라 일제히 콧소리로 ‘허밍’하고, 다시 반대 측이 ‘허밍’하면 찬반 양측의 소리 크기 차에 따라 어느 쪽이 우세한지 판단하는 것이다. 허밍은 직접적인 투표는 아니면서 다소간의 익명성을 보장하며 대략적인 합의를 살필 수 있기 때문에 사용되며, 워킹그룹에 따라 허밍이란 제목의 이메일로 찬반 상태를 파악하는 경우도 있다. 허밍은 건조한 합의 과정에 재미를 주는 요소이기도 하다.
- 15 역자주\_인터넷 표준을 만들어 가는 과정은 RFC 1602(<http://www.rfc-editor.org/info/rfc1602>)에 자세하게 설명되어 있다.
- 16 역자주\_‘선 비행, 후 구매’라는 표현은 미군이 도입한 전략 무기 구입 정책의 변화에서 유래한 말이다. 무기 구매 계획은 대부분 실제 테스트 없이 서류 검토만으로 이루어졌지만 이제는 대량의 항공기를 구매하기 전에 반드시 사전 테스트를 거쳐 기종을 선정한다. 이러한 ‘선 비행, 후 구매’ 정책은 표준화 과정에서 실제 구현이 이루어지도록 하는 IETF의 ‘작동하는 코드(running code)’ 원칙과 일맥상통한다.
- 17 역자주\_참조 구현(reference implementation)이란, 제안한 표준을 시험하기 위해 완성된 형태의 실체로 구현하는 것이다. 예를 들어 CGI 인터페이스에 대한 새로운 표준을 제안하면, 다양한 종류의 구현 방법이 제시되어 표준화 과정에서 이들을 포괄할 수 있는 다중 구현(multiple implementation)으로 통합되는데, 아파치 서버와 리눅스와 같은 특정 플랫폼에서 구동할 수 있게 만든 CGI 검색 프로그램은 다중 구현으로 인정 할 수 있는 참조 구현이 된다.

## 4 | GNU 운영체제와 자유 소프트웨어 운동

리처드 스톤먼 Richard Stallman

송창훈 역

### 최초의 소프트웨어 공유 공동체

MIT 인공지능연구소<sup>01</sup>에서 일하기 시작한 1971년부터, 나는 그곳에 존재했던 소프트웨어 공유 공동체의 일원이 되었다. 소프트웨어 공유가 그곳만의 문화는 아니었다. 요리법을 공유하는 것이 요리의 역사만큼이나 오래된 것처럼, 소프트웨어를 공유하는 것도 컴퓨터만큼 오래된 것이다. 그러나 우리는 더욱 이상적인 우리만의 공동체를 만들어 갔다.

MIT 인공지능연구소는 시분할 운영체제 ITS<sup>Incompatible Time-sharing System</sup>를 사용했는데, 이것은 연구소 해커<sup>02</sup>들이 당시 가장 큰 시스템의 하나였던 DEC의 PDP-10<sup>03</sup>에 탑재할 목적으로 어셈블리 언어로 개발한 것이었다. 공동체의 일원이자 연구소 시스템 해커로서의 내 임무는 ITS를 개량하는 것이었다.

당시에는 ‘자유 소프트웨어’라는 용어가 아직 존재하지 않았기 때문에 우리가 만든 소프트웨어를 그렇게 부르진 않았지만, 본질은 지금의 ‘자유 소프트웨어’ 바로 그것이었다. 다른 대학이나 기업 사람들이 우리 프로그램을 이식해 사용하고 싶어할 때면, 우리는 기꺼이 그렇게 하도록 했다. 만약 여러분이 낯설고 흥미로운 프로그램을 사용하는 사람을 보았다면, 언제든 소스 코드를 요청해 읽고 수정하거나 일부를 떼어내 새로운 프로그램을 만드는 데 사용할 수 있었다.

## 공동체의 붕괴

DEC가 PDP-10 시리즈의 생산을 중단한 1980년대 초부터 상황은 급격히 변해갔다. 1960년대에는 강력하고 우아한 설계였지만, 태생적으로 1980년대 들어 실현 가능해진 더 큰 메모리 주소 공간을 사용할 수 있게 확장될 수는 없었다. 이는 ITS를 구성하는 거의 모든 프로그램이 쓸모없게 된다는 것을 의미했다.

인공지능연구소의 해커 공동체도 오래지 않아 붕괴되었다. 1981년에는 스피노프 회사<sup>04</sup> 심벌릭스Symbolics가 연구소 해커의 대부분을 계속 빼내갔고, 구성원이 없어진 공동체는 자체적으로 유지될 수 없었다(스티븐 레비Steven Levy의 책『해커, 그 광기와 비밀의 기록』<sup>05</sup>에는 이러한 일들이 전성기 때의 공동체 일화와 함께 자세히 묘사되어 있다). 인공지능연구소가 새로 PDP-10을 구입한 1982년에는 운영자들이 ITS가 아닌 DEC의 비자유<sup>06</sup> 시분할 운영체제를 사용하기로 결정했다.

VAX백스나 68020<sup>07</sup>과 같은 당시의 현대적 컴퓨터에는 전용 운영체제가 탑재되었는데, 이들은 모두 자유 소프트웨어가 아니었다. 따라서 실행 파일 복제물을 얻을 때조차 기밀유지계약에 서명해야만 했다.

이것은 컴퓨터를 사용하는 첫 단계가 주위 사람을 돋지 않겠다고 약속하는 것이라는 의미였다. 협력적인 공동체는 금지되었다. 자유 소프트웨어proprietary software<sup>08</sup> 소유자들이 만든 규칙은, “이웃과 공유하는 자는 해적이다.”<sup>09</sup> 소프트웨어에 수정이 필요하면, 우리에게 만들어 달라고 사정하라”였다.

자유 소프트웨어 제도의 개념, 즉 ‘소프트웨어를 공유하거나 수정하는 것을 허용하지 않는 사회제도’에 대해 반사회적이고 비윤리적이며, 그야말로 잘못된 것이라고 말한다면 어떤 독자는 놀랍게 느낄지도 모른다. 그러나 사람들 사이를 갈라놓고 사용자를 무력하게 만드는 것에 기초한 제도에 대해 다르게 뭐라 이야기할 수 있겠는가? 만약 이러한 견해에 생소한 독자가 있다면, 자유 소프트웨어 사회제도를 본래

부터 주어진 것으로 의심 없이 받아들였거나, 자유 소프트웨어 산업이 원하는 대로 내놓은 말에 따라 판단한 것이라 할 수 있다. 소프트웨어 발행사<sup>10</sup>들은 이 문제를 바라보는 관점이 오직 하나밖에 없다고 사람들이 믿게 하기 위해 오랫동안 노력해 왔다.

소프트웨어 발행사들이 말하는 “저작권 침해를 중단하라!”나 “우리의 권리를 행사 한다!”는 등의 표현은 부차적인 것이고 그 안에는 그들이 당연하다고 생각하는 억측이 숨어 있다. 그들은 법률적이고 강압적인 표현을 통해 대중이 비판 없이 억측을 받아들이게 하고 있는 것이다. 이제 그들의 억측을 검토해 보자.

첫 번째 억측은 소프트웨어 회사가 소프트웨어를 소유할 수 있는 의심할 의지 없는 자연권을 가졌기 때문에 모든 소프트웨어 사용자를 지배할 수 있다는 것이다. 만약 그것이 자연권이라면, 어떠한 해를 끼쳐도 우리는 대항할 수 없다. 그러나 흥미롭게도, 미국의 헌법과 법률 전통은 이러한 관점을 거부한다. 저작권은 자연권이 아니다. 단지 연방정부가 인위적으로 부여한 독점권에 의해 사용자의 복제 자연권이 제한되는 것이다.

두 번째 억측은 소프트웨어에 있어 중요한 것은 오직 작업이지, 그를 통해 만들어지는 사회가 아니라는 것이다. 즉 컴퓨터 사용자는 소프트웨어로 할 수 있게 허용된 일에만 신경 써야 할 뿐, 그들이 가질 수 있는 사회에 대해 관심을 가져서는 안 된다는 것이다.

세 번째 억측은 소프트웨어 회사에 프로그램 사용자에 대한 지배력을 주지 않으면, 쓸 수 있는 소프트웨어나 특정 작업에 필요한 소프트웨어를 구하지 못하게 된다는 것이다. 이 억측은 자유 소프트웨어 운동이 그렇게 하지 않고도 많은 유용한 소프트웨어를 만들 수 있다는 것을 증명하기 전까지 타당할 수 있었다.

만약 우리가 이런 억측들을 거부하고 사용자를 최우선으로 고려한 일반적인 상식과 윤리에 따라 판단한다면, 매우 다른 결론에 도달하게 된다. 다른 사람을 돋는 것은 사회의 근간이므로, 컴퓨터 사용자는 자신의 필요에 맞춰 프로그램을 자유롭게 고칠 수 있어야 하고 또한 자유롭게 공유할 수 있어야 한다.

이러한 결론이 도출되는 자세한 설명은 지면 관계상 생략하지만, 독자들이 ‘왜 소프트웨어는 소유자가 있으면 안 되는가(<http://www.gnu.org/philosophy/why-free.ko.html>)’라는 글을 참고했으면 한다.

## 피할 수 없는 도덕적 선택

공동체가 사라지면서 예전처럼 공유 활동을 계속하는 것이 불가능했고, 대신 나는 피할 수 없는 도덕적 선택의 기로에 섰다.

손쉬운 선택은 기밀유지계약에 서명하고 동료 해커를 돋지 않겠다고 약속하면서 사유 소프트웨어 세계에 합류하는 것이었다. 그렇게 했다면 내가 개발했을 소프트웨어 또한 기밀유지계약에 따라 배포됐을 것이므로, 다른 사람이 동료를 돋지 못하도록 강요하는 데 가세했을 것이다.

나는 그런 방법으로 돈을 벌 수 있었을 것이며, 아마도 프로그램을 만드는 즐거움도 누릴 수 있었을 것이다. 그러나 훗날 인생을 정리하며 과거를 돌아켜 봤을 때, 내가 사람들 사이를 단절시키는 벽을 만들었으며 또한 이 세상을 더 나쁘게 만들어 왔다고 느끼게 되리라는 점도 알고 있었다.

나는 인공지능연구소와 나에게 제어 프로그램의 소스 코드를 주기 거부했던 사람 때문에 프린터 사용에 곤란을 겪은 경험이 있었다. 즉, 제어 프로그램에 특정 기능이 빠져 있어 프린터 사용이 몹시 불편했던 상황을 이미 겪었기 때문에 기밀유지계약이 가져올 결과를 이미 알고 있었던 것이다.

그래서 나는 기밀유지계약이 무해하다고 말할 수 없다. 또한 그가 소스 코드를 공유하기 거부했을 때 나는 매우 분개했었기 때문에, 입장은 바꿔 다른 사람에게 같은 일을 할 수는 없었다.<sup>11</sup>

간단하지만 유쾌하지 않은 다른 선택은, 컴퓨터 분야를 떠나는 것이었다. 이 선택에서는 내가 가진 기술이 악용되지 않겠지만, 또한 내 기술이 버려지는 것이라 할 수 있었다. 컴퓨터 사용자를 분리하고 제한한 비난이 내게 돌아오진 않겠지만, 그런 일들은 어쨌든 일어나게 될 것이었다.

그래서 나는 프로그래머로서 뭔가 좋은 일을 할 수 있는 방법을 찾게 되었다. 나는 내 자신에게 물어보았다. 공동체를 다시 부활시키기 위해 내가 만들 수 있는 프로그램은 없을까?

해답은 명확했다. 제일 먼저 필요한 프로그램은 운영체제였다. 운영체제는 컴퓨터를 사용할 수 있게 하는 결정적인 소프트웨어다. 운영체제가 있으면 많은 것을 할 수 있지만, 그렇지 않으면 컴퓨터 자체를 구동할 수 없다. 자유 운영체제가 있다면 다시 한번 협력적인 해커 공동체를 가질 수 있고 누구에게나 합류를 권할 수 있다. 동료의 자유를 빼앗는 기밀유지계약으로 시작하지 않아도 누구든 컴퓨터를 사용할 수 있을 것이다.

운영체제 개발자로서 나는 이 일에 딱 맞는 기술을 갖고 있었다. 그래서 실패할 가능성이 당연히 있음에도, 이 일을 내 소명으로 받아들였다. 나는 새로운 운영체제를 유닉스 Unix와 호환되게 만들기로 했다. 그렇게 하면 이식성을 갖게 되고 기존의 유닉스 사용자가 쉽게 넘어올 수 있을 것으로 생각했다. 이름은 해커 전통에 따라 GNU<sup>그누</sup><sup>12</sup>라 지었다. 이것은 ‘GNU's Not Unix’라는 뜻의 재귀적 두문자어<sup>13</sup>다.

운영체제는 다른 프로그램을 실행시키는 커널 kernel<sup>14</sup>만을 의미하지 않는다. 1970년대의 모든 운영체제는 그 이름에 걸맞게 명령처리기, 어셈블러, 컴파일러, 인터

프리터, 디버거, 텍스트 편집기, 메일 프로그램 그리고 더 많은 것들을 포함했다. ITS와 멀티스<sup>Multics</sup>, VMS 그리고 유닉스도 모두 이런 프로그램들을 갖고 있었다.<sup>15</sup> 따라서 GNU 역시 이런 프로그램들을 갖춰야 했다.

훗날 나는 힐렐<sup>16</sup>이 남긴 명언을 알게 됐는데, GNU 프로젝트를 시작하게 된 결단은 이러한 정신과 맥을 같이 한다고 볼 수 있다.

스스로 소중히 여기지 않는데, 당신을 위해 줄 사람이 있을까?

자신만을 생각한다면, 온전한 한 인간이 될 수 있을까?

바로 지금이 아니면, 시작할 때가 다시 있을까?

## 구속되지 않는다는 관점에서의 자유

자유 소프트웨어라는 용어는 때때로 무료라는 뜻으로 잘못 이해되는데, 사실 자유 소프트웨어는 금전적인 측면과 전혀 관계가 없고 자유에 대한 관점과 관계가 있다.

자유 소프트웨어의 정의는 다음과 같다.

- 어떤 목적이든 원하는 대로 프로그램을 실행시킬 수 있는 자유
- 무료 또는 유료로 프로그램 복제물을 재배포할 수 있는 자유
- 필요에 따라 프로그램을 개작할 수 있는 자유(이 자유가 실제로 보장되기 위해서는 소스 코드를 이용할 수 있어야 한다. 왜냐하면 소스 코드 없이 프로그램을 개작하는 것은 극도로 어렵기 때문이다)
- 공동체 전체가 개선된 이익을 나눌 수 있게, 개작한 프로그램을 배포할 수 있는 자유

'자유'라는 단어는 금전적인 측면이 아닌 구속되지 않는다는 관점에서의 자유를 의미하기 때문에 자유 소프트웨어를 유료로 판매하는 데는 어떠한 모순도 존재하지 않는다. 실제로 소프트웨어를 판매할 수 있는 자유는 매우 중요하다. CD-ROM에

담겨 판매되는 자유 소프트웨어 모음은 공동체를 위해 필요하며, 자유 소프트웨어 개발 기금 마련을 위한 중요한 수단이기도 하다. 따라서 이러한 소프트웨어 모음, 즉 자유 소프트웨어 수집 저작물에 자유롭게 포함시킬 수 없는 프로그램은 자유 소프트웨어가 아니라고 할 수 있다.

'자유'라는 단어가 가진 모호함 때문에 사람들은 오랫동안 대체 용어를 찾아왔지만, 그 누구도 더 나은 용어를 찾지 못했다. 영어는 다른 언어에 비해 많은 단어와 뉘앙스를 갖고 있지만 단순함과 명료함을 갖고 있지는 않다. 자유 소프트웨어에 사용된 단어 free는 unfettered, 즉 구속에서 벗어났다는 의미에 가장 가깝다. liberated나 freedom, open과 같은 단어는 원하는 의도를 표현하는 데 불리하거나 잘못된 의미를 갖고 있다.<sup>17</sup>

## GNU 소프트웨어와 GNU 시스템

완전한 운영체제 전체를 개발하는 것은 매우 큰 프로젝트다. 능력에 맞게 할 수 있는 일에 다가서기 위해서 나는 이미 존재하는 자유 소프트웨어를 적용 가능한 곳에 활용하기로 했다. 예를 들면 맨 처음에는 기본 조판 프로그램으로 텍TeX<sup>18</sup>을 사용하기로 했고, 몇 년 뒤에는 새로운 윈도 시스템을 개발하는 대신 X 윈도 시스템을 GNU에 사용하기로 했다.

이런 결정과 다른 비슷한 결정들 덕분에 GNU 시스템은 단순히 GNU 소프트웨어 만을 모아놓은 것과는 다른 것이 되었다. GNU 시스템에는 우리가 개발한 것뿐만 아니라 다른 사람과 다른 프로젝트가 나름의 목적을 갖고 개발한 프로그램도 포함되는데, 이것은 그 프로그램들이 자유 소프트웨어여서 한데 모으는 것이 가능했다.

## 프로젝트의 시작

1984년 1월, 나는 MIT의 일을 그만두고 GNU 소프트웨어를 만들기 시작했다. MIT 직원으로 남아 있게 되면, 업무상 저작으로 간주해 MIT가 내 작업물에 대한

소유권을 주장하거나 자체 배포 규정을 적용할 수 있으며, 심할 경우 자유 소프트웨어로 바꿀 수도 있었다. 따라서 GNU 소프트웨어를 자유 소프트웨어로 배포하는 데 방해받지 않으려면, MIT를 그만둘 필요가 있었다. 나는 새로운 소프트웨어 공유 공동체를 만들기 위한 많은 작업이, 쓸모없게 되어버리는 것을 원치 않았다.

그렇지만 당시 인공지능연구소장이던 윈스턴 Patrick H. Winston 교수는 친절하게도 내가 연구실 장비를 계속 사용할 수 있게 배려해 주었다.

## 첫 번째 단계

GNU 프로젝트를 시작하기 얼마 전에 나는 VUCK라는 이름으로도 불리던 자유 대학 컴파일러 키트 Free University Compiler Kit를 알게 되었다(자유를 의미하는 네덜란드어 단어는 v로 시작된다).<sup>19</sup> 이 컴파일러는 C와 파스칼을 포함한 여러 언어를 처리할 수 있었고 다양한 타깃 머신<sup>20</sup>을 지원했다. 나는 VUCK 개발자에게 그것을 GNU에 사용해도 될지 묻는 메일을 보냈다.

그는 대학은 자유롭지만, 컴파일러는 그렇지 않다면 비웃는 듯한 회신을 보내왔다. 그래서 나는 첫 번째 GNU 프로젝트 프로그램으로 여러 언어 및 플랫폼을 지원하는 컴파일러를 만들 결심을 했다.

컴파일러 전체를 직접 만들어야 하는 수고를 피할 수 있으리란 희망으로, 나는 로렌스 리버모어 연구소 Lawrence Livermore National Laboratory가 개발한 파스텔 Pastel 컴파일러의 소스 코드를 입수했다. 이 컴파일러는 멀티플랫폼을 지원했으며, 시스템 프로그래밍 언어로 사용할 목적으로 파스칼 언어를 확장한 파스텔 언어로 만든 것이다. 물론 파스텔 컴파일러는 파스텔 언어를 컴파일하는 데 사용하는 것이다. 나는 C 언어 프런트 엔드 front-end를 덧붙여 모토로라 68000 컴퓨터로 이식하기 시작했다. 그러나 파스텔 컴파일러는 수 MB 용량의 스택을 요구하지만, 68000 기반 유닉스 시스템에서는 단지 64KB밖에 이용할 수 없다는 것을 알게 되었을 때 나는 계획을 포기할 수밖에 없었다.

파스텔 컴파일러는 입력 파일 전체를 파싱해 하나의 신팩스 트리로 구성한 다음, 이를 연속된 명령으로 변환한 뒤 전체 출력 파일을 생성하는 구조였고, 그 과정에서 어떠한 기억공간 반환도 일어나지 않는다.

그래서 나는 새로운 컴파일러를 처음부터 다시 만들어야 한다는 결론을 내렸다.<sup>21</sup> 파스텔 컴파일러를 포기하고 새로 개발한 컴파일러는 이제 GCC<sup>22</sup>라는 이름으로 불리고 있다. GCC에는 파스텔 컴파일러의 소스 코드를 전혀 사용하지 않았지만, 파스텔에 추가했던 C 언어 프린트 엔드는 GCC에도 계속 사용하려고 했다. 그러나 이러한 일들은 모두 몇 년 뒤에 이루어진 것이었고, 당시에는 우선 GNU 이맥스 개발에 집중했다.

## GNU 이맥스

1984년 9월부터 만들기 시작한 GNU 이맥스<sup>Emacs<sup>23</sup></sup>는 1985년 초에 제법 쓸만한 상태가 되었다. 나는 유닉스 텍스트 편집기인 vi나 ed에 흥미가 없었기 때문에 다른 기종에서 편집 작업을 했는데, 이맥스를 만든 뒤에는 유닉스 시스템에서 작업할 수 있었다.

그 시기에 GNU 이맥스를 사용하고 싶어하는 사람들이 생겨나 배포 방법을 고민하게 됐다. 물론 내가 사용하던 컴퓨터의 익명 FTP 서버에는 올려두었다(이 컴퓨터의 이름은 prep.ai.mit.edu<sup>24</sup>였으며, GNU의 주된 FTP 배포 사이트가 되었다. 몇 년 뒤에 이 컴퓨터를 퇴역시킨 후에도 이름은 새로운 FTP 서버로 계속 이어갔다). 하지만 당시에는 인터넷을 이용할 수 있는 사람이 많지 않아 FTP를 통한 다운로드가 어려웠다. 그렇다면 이맥스를 원하는 사람들에게 뭐라고 말해 줘야 할 것인가?

나는 인터넷을 통해 다운로드할 수 있는 친구를 찾아보라고 했다. 또한 테이프<sup>25</sup>를 반송용 봉투와 함께 우편으로 보내면, 내가 사용하던 PDP-10용 이맥스를 복제해 보내주겠다고 했다. 당시 나는 직장이 없었기 때문에 자유 소프트웨어로 수입을 얻

을 방법을 찾고 있었고, 150달러<sup>26</sup>를 지불하면 누구에게나 이맥스 테이프를 우송해 주는 방법을 생각해냈다. 나는 그렇게 자유 소프트웨어 배포 사업을 시작했고, 오늘날 GNU/리눅스 시스템 전체 배포판을 판매하는 많은 회사의 시초가 되었다.

## 프로그램은 누구에게나 자유로운가?

원저작자의 손을 떠날 때 자유 소프트웨어인 프로그램이라 하더라도, 그 프로그램이 모든 사람에게 항상 자유 소프트웨어로 남게 되지는 않는다. 예를 들면 저작권이 없는 공중영역 소프트웨어public domain software는 자유 소프트웨어다. 하지만 누구든 그것을 개작해 자유 소프트웨어로 만들 수 있다. 마찬가지로 소프트웨어를 개작한 후에 자유 소프트웨어로 만들 수 있는 ‘허용적 이용허락permissive license’<sup>27</sup>으로 배포되는, 저작권이 있는 자유 프로그램도 많다.

이러한 문제의 전형적인 예가 X 윈도 시스템이다. MIT가 개발한 X 윈도 시스템은 자유 소프트웨어로 배포되었지만, 허용적 이용허락이 적용되었기 때문에 이내 많은 컴퓨터 회사가 X 윈도를 차용하게 됐다. 그들은 X 윈도 시스템을 그들의 자유 유닉스 시스템에 바이너리 형태로만 추가한 뒤에 기존의 유닉스와 동일한 기밀유지계약 아래 두었다. 이러한 종류의 X 윈도 시스템은 유닉스와 마찬가지로 더 이상 자유 소프트웨어가 아니다.

X 윈도 시스템 개발자들은 이런 문제를 고려하지 않았으며, 오히려 그렇게 되기를 의도하고 기대했다. 왜냐하면 그들의 목적은 ‘자유’가 아니라 단지 ‘많은 사용자를 가진’ 것으로 정의할 수 있는 ‘성공’이었기 때문이다. 그들은 사용자가 자유를 갖게 되는가에 상관없이 오직 많은 수의 사용자를 갖게 되는 데만 관심을 가졌다.

이것은 “이 프로그램은 자유로운가?”라는 질문에 다른 답이 도출되는, 자유의 총량을 산정하는 데 두 가지 다른 기준이 존재하는 역설적인 상황을 초래했다. 만약 MIT 배포 기준에 따라 자유를 판단한다면 X 윈도 시스템은 자유 소프트웨어라고

할 수 있지만, X 윈도 일반 사용자의 자유를 기준으로 판단한다면 자유 소프트웨어라고 해야만 한다. 대부분의 X 윈도 사용자는 자유 소프트웨어가 아닌, 유닉스 시스템에 딸려 오는 자유 소프트웨어를 사용하는 것이다.

## 카피레프트와 GNU GPL

GNU의 목적은 인기를 얻는 것이 아닌 사용자에게 자유를 주는 것이다. 그래서 우리는 GNU 소프트웨어가 자유 소프트웨어로 바뀌는 것을 막는 배포 규정을 만들어야 했다. 우리가 사용한 방법을 카피레프트<sup>28</sup>라 부른다.

카피레프트는 저작권법을 이용한다. 그러나 저작권법의 주된 목적을 반대로 이용해서, 프로그램을 제한하는 대신 프로그램을 자유롭게 유지하는 수단으로 삼는다.

카피레프트의 핵심 개념은 모든 사람이 프로그램을 실행하고 복제, 개작하며 또한 개작한 프로그램을 배포할 수 있게 허용하는 것이다. 그러나 사용자 자신이 별도로 만든 제한을 추가하는 것은 허용되지 않는다. 즉 ‘자유 소프트웨어’를 규정하는 결정적인 자유를 모든 사람에게 보장하는 것이다. 이것은 양도할 수 없는 권리가 된다.<sup>29</sup>

효과적인 카피레프트를 위해서는 개작된 프로그램도 자유로워야만 한다. 이것은 우리가 만든 원저작물에 기반한 2차적 저작물이 공개되었을 때, 그것이 우리 공동체로 다시 돌아올 수 있게 보장한다. 직업 프로그래머가 GNU 소프트웨어를 개선하는 작업에 자원했을 때, 고용주가 작업물을 빼앗아 자유 소프트웨어로 만들려는 시도를 가로막는 것도 카피레프트다.

프로그램 사용자 모두의 자유를 보장하려면, 개작된 부분도 자유로워야 한다는 것이 필수적이다. X 윈도 시스템을 사유화한 회사들은 자사의 시스템과 하드웨어에 이식하기 위해 보통 약간의 개작을 한다. 이런 개작 부분은 X 윈도 전체와 비교하면 작지만, 하찮은 것은 아니다. 만약 개작이 사용자의 자유를 거부하는 변명이 될 수 있다면, 이런 변명의 이점을 누구나 쉽게 이용할 수 있을 것이다.

관련 쟁점으로 비자유 코드를 자유 프로그램에 결합하는 문제가 있다. 이러한 결합의 결과는 필연적으로 비자유가 된다. 비자유 부분이 가진 자유의 결여가 결합된 전체로 확대되는 것이다.<sup>30</sup> 이러한 결합을 허용하는 것은 배를 침몰시키기 충분한 큰 구멍을 만든다. 따라서 카피레프트를 위한 결정적 요건은 이런 구멍을 막는 것이다. 카피레프트 프로그램에 덧붙이거나 결합하는 그 어떤 것도, 결합 후에 전체가 다시 자유로운 카피레프트가 되는 것이어야 한다.

대부분의 GNU 소프트웨어에 적용할 수 있게 카피레프트를 실제로 구현한 것이 GNU 일반 공중 이용허락(GNU General Public License), 즉 GNU GPL이다. 특정 상황에서 사용할 수 있는 다른 카피레프트도 있다.<sup>31</sup> GNU 매뉴얼에도 역시 카피레프트를 사용하는데, 소프트웨어의 특성에 맞춘 GPL의 복잡성이 매뉴얼에는 적합하지 않기 때문에 좀 더 간소한 형태의 카피레프트를 사용한다.<sup>32</sup>

## 자유 소프트웨어 재단

이맥스 사용에 대한 관심이 높아지면서 GNU 프로젝트에 참여하는 새로운 사람들 이 생겨났고, 우리는 다시 한번 기금을 모을 때가 왔다고 판단했다. 그래서 1985년 자유 소프트웨어 개발을 지원할 비과세 자선단체인 자유 소프트웨어 재단(Free Software Foundation)을 설립했다.<sup>33</sup> FSF는 이맥스 테이프 배포 사업을 넘겨받았고, GNU뿐 아니라 다른 자유 소프트웨어 테이프 배포로 사업 범위를 넓히며 자유 매뉴얼 판매 또한 병행했다.

FSF 수입의 대부분은 (개작과 재배포의 자유가 보장되는 소스 코드 CD-ROM, 바이너리 CD-ROM, 고품질 인쇄 매뉴얼 그리고 전체 소프트웨어를 소비자가 선택한 플랫폼에 맞춰 넣은 딜럭스 배포판 등의) 자유 소프트웨어 판매와 관련 서비스로 들어왔다. 지금은 <http://shop.fsf.org/>를 통해 매뉴얼과 다른 여러 물품을 판매하지만, 후원 회비가 수입의 큰 부분을 차지한다. <http://fsf.org/join>을 통해 FSF에 참여할 수 있다.

자유 소프트웨어 재단이 고용한 사람들은 많은 종류의 GNU 소프트웨어 패키지를 만들고 관리해왔다. 그 중 유명한 것으로 C 라이브러리와 셸shell이 있다. 롤런드 맥그래스Roland McGrath가 개발한 GNU C 라이브러리는 GNU/리눅스 시스템에서 실행되는 모든 프로그램이 리눅스와 통신할 때 사용된다. 브라이언 폭스Brian Fox는 대부분의 GNU/리눅스 시스템에서 셸로 사용되는 배시Bash<sup>34</sup>를 개발했다.

우리는 이런 프로그램들을 개발하기 위해 기금을 사용했는데, 그것은 GNU 프로젝트가 단지 도구나 개발 환경에 대한 것만이 아니기 때문이다. 우리의 목표는 완전한 운영체제를 만드는 것이었고, 이러한 프로그램들은 그 목적을 위해 필요했다.

## 자유 소프트웨어 지원

자유 소프트웨어 철학은 널리 만연된 특정한 사업 관행을 거부한다. 그러나 이것이 사업 자체에 대한 반대는 아니다. 사용자의 자유를 존중한다면 우리는 그들의 성공을 기원할 것이다.

이맥스 판매는 자유 소프트웨어 사업의 한 예를 보여준다. FSF로 이 사업이 이관됐을 때, 나는 생계를 위해 또 다른 일이 필요했는데, 내가 개발한 자유 소프트웨어에 대한 서비스 판매에서 활로를 찾을 수 있었다. 여기에는 이맥스 프로그래밍이나 GCC 맞춤변경 등과 같은 교육이 포함됐지만, 대부분은 GCC를 새로운 플랫폼으로 이식하는 소프트웨어 개발이었다.

지금은 이 같은 자유 소프트웨어 사업을 많은 회사가 한다. 어떤 회사는 자유 소프트웨어 수집물을 CD-ROM으로 배포하고, 또 다른 회사는 사용자의 질문에 답해주는 것부터 소프트웨어 오류 수정이나 중요 기능을 새롭게 추가해 주는 것까지 다양한 서비스 지원을 판매한다. 심지어 새로운 자유 소프트웨어 제품을 시장에 출시하는 것을 목표로 하는 자유 소프트웨어 회사도 생겨나고 있다.

그러나 ‘오픈 소스’라는 용어로 홍보하는 많은 회사가, 실제로는 단지 자유 소프트웨어와 연동될 뿐인 비자유 소프트웨어 사업을 한다는 사실에 주의해야 한다. 그들은 자유 소프트웨어 회사가 아니다. 사용자를 자유로부터 멀어지게 유혹하는 제품을 판매하는 자유 소프트웨어 회사다. 그들이 ‘부가가치 패키지’라고 부르며 우리가 받아들이길 원하는 가치는 자유보다 편리함인데, 우리가 자유에 더 큰 가치를 부여한다면 그것을 ‘가치를 더한 것’이 아닌 ‘자유를 빼낸’ 패키지라 불러야 마땅하다.

## 기술 목표

GNU의 주된 목표는 자유다. GNU가 유닉스보다 나은 기술적 이점을 가질 수 없다 해도, 사용자가 서로 협력하는 사회적 이점과 사용자의 자유를 존중하는 윤리적 이점은 가질 수 있다.

그러나 좋은 기준으로 알려진 표준을 우리 작업에 적용한 것은 자연스러운 일이었다. 예를 들면 자료구조에 대한 고정 크기 임의 제한을 막기 위해 동적 할당을 택한 것이나, 적합한 것은 모두 8비트 코드로 처리한 것 등이다.<sup>35</sup>

또한 16비트 기계를 지원하지 않기로 함으로써, 유닉스가 집중하던 작은 메모리 문제에서 벗어나 1MB를 넘지 않는 한, 메모리 사용을 줄이기 위해 노력할 필요가 없어졌다(GNU 시스템이 완성될 시점에는 32비트 기계가 표준이 될 것이 확실했다). 따라서 매우 큰 용량의 파일을 다루는 경우가 아니라면, 프로그래머에게 입출력에 대한 문제를 고려하지 말고 입력 파일 전체를 메모리로 올려 사용하도록 권장 할 수 있었다.<sup>36</sup>

이러한 결정은 속도와 신뢰성에 있어 많은 GNU 프로그램이 그에 대응하는 유닉스 프로그램을 능가할 수 있게 했다.

## 기증받은 컴퓨터

GNU 프로젝트의 명성이 높아지면서 유닉스가 탑재된 시스템을 기증하는 사람도 생겨났다. GNU를 구성하는 프로그램을 가장 쉽게 개발하는 방법은 유닉스 환경에서 먼저 구현한 뒤에 이를 GNU에 맞게 하나씩 바꿔가는 것이다. 따라서 유닉스 시스템은 우리에게 매우 유용했다. 그러나 이것은 우리가 유닉스를 사용하는 것을 바르냐라는 윤리적 문제를 불러왔다.

유닉스는 예나 지금이나 사유 소프트웨어다. 그리고 GNU 프로젝트의 철학은 사유 소프트웨어를 사용하지 않겠다는 데 있다. 그렇다면 사유 소프트웨어인 유닉스를 이용해서 자유 소프트웨어를 만드는 것은 모순이 아니냐는 문제가 제기된다. 내가 내린 결론은 폭력에 대한 자기방어가 정당화될 수 있는 것과 같은 논리로, 사유 소프트웨어를 막을 자유 대체물 개발에 결정적으로 필요하다면 사유 소프트웨어의 사용도 정당화될 수 있다는 것이다.

그러나 필요악이라 하더라도, 나쁜 것은 어쩔 수 없는 사실이다. 지금은 자유 운영체제로 유닉스 대체를 완료했기 때문에 더 이상 어떠한 유닉스 복제물도 갖고 있지 않다. 만약 우리가 컴퓨터의 운영체제를 자유 운영체제로 대체하지 못했다면, 컴퓨터를 자유 운영체제를 실행할 수 있는 다른 것으로 교체했을 것이다.

## GNU 태스크 리스트

프로젝트 진행 과정에서 발견하거나 개발한 시스템 구성 요소가 늘어나면서 남은 작업을 목록화하는 것이 편리할 때가 왔다. 이 목록은 GNU 태스크 리스트<sup>GNU Task List</sup>로 알려졌고, 필요한 작업을 위한 개발자 모집에 이용됐다. 더 만들어야 할 유닉스 대체물과 함께 우리는 진정으로 완전한 운영체제가 갖춰야 할 것으로 생각한다양하고 유용한 소프트웨어와 문서 프로젝트를 이 목록에 추가했다.

지금<sup>37</sup>은 GNU 태스크 리스트에 남은 유닉스 구성 대체물이 거의 없다. 없어도 무방한 몇몇을 빼면 개발이 모두 완료되었다. 그러나 ‘애플리케이션’이라 부를 수 있을 많은 소프트웨어 프로젝트가 이 목록을 채우고 있다. 일정 수 이상의 사용자가 흥미를 갖는 프로그램이라면 운영체제에 추가할 만한 유익한 것이 될 수 있다.

심지어 게임도 처음부터 목록에 들어갔는데, 유닉스에는 게임이 포함되기 때문에 GNU도 자연스럽게 그렇게 되었다. 그러나 게임에는 별다른 호환성 문제가 없어 유닉스의 게임 종류를 그대로 따르지 않고 사용자가 좋아할 만한 것으로 다양하게 포함시켰다.

## GNU 라이브러리 GPL

GNU C 라이브러리에는 사유 소프트웨어와의 링크를 허용하는 특별한 종류의 카피레프트인 GNU 라이브러리 일반 공중 이용허락, 즉 LGPL<sup>38</sup>이 사용된다. 왜 이런 예외를 만들었을까?

이것은 원칙이 아닌 전략의 문제다. 사유 소프트웨어 제품에 우리 코드를 사용하게 해주는 원칙은 없다. (우리와의 공유를 거부할 것이 뻔한 프로젝트에 기여할 이유가 무엇인가?) C 라이브러리나 기타 다른 라이브러리에 LGPL을 사용하는 것은 일종의 전략이다.

C 라이브러리는 포괄적인 기능을 하며, 모든 사유 운영체제와 컴파일러에 포함되어 있다. 따라서 사유 소프트웨어가 GNU C 라이브러리를 사용할 수 없어도 다른 라이브러리가 있으므로 별다른 문제가 되지 않는다. 이 경우에는, 사유 소프트웨어에 해는 없지만 더해지는 이점도 없고, GNU C 라이브러리의 활용은 장려되지 않을 것이다.

그러나 GNU/리눅스를 비롯한 GNU 시스템에는 C 라이브러리로 오직 GNU C 라이브러리만이 존재한다. 따라서 GNU C 라이브러리의 배포 규정에 의해 GNU 시

스템에서 작동하는 사유 프로그램을 만들 수 있을지 여부가 결정된다. 사유 애플리케이션이 GNU 시스템에서 실행되도록 허용할 윤리적인 이유는 없다. 그러나 전략적으로 볼 때, 이것을 허용하지 않으면 자유 애플리케이션 개발을 장려하는 것 보다 GNU 시스템의 활용을 저하시킬 가능성이 높다. 이것이 LGPL이 C 라이브러리에 대한 좋은 전략이 되는 이유다.<sup>39</sup>

다른 라이브러리에 대해서는 각각의 상황을 고려한 전략적 결정이 필요하다. 라이브러리가 특정한 범주의 프로그램을 작성하는 데 도움을 주는 특별한 기능을 수행하는 경우라면 GPL로 배포해 자유 프로그램들만 사용할 수 있게 해서 사유 소프트웨어에 맞서는 이점을 주면 자유 소프트웨어 개발자를 돋는 방법이 된다.

GNU 리드라인<sup>Readline</sup>의 예를 보자. 이 라이브러리는 배시에 명령행 편집 기능<sup>40</sup>을 제공하기 위해 개발되었고, LGPL이 아닌 GPL로 배포된다. 이 때문에 리드라인의 사용 총량은 줄었을지 몰라도 우리가 입은 손실은 없다. 반면에 리드라인을 통해 최소한 하나의 유용한 애플리케이션이라도 자유 소프트웨어로 만들어졌다면, 그것은 공동체를 위한 실제 이익이 된다.

사유 소프트웨어 개발자들은 자본이 주는 이점을 갖고 있다. 그러나 자유 소프트웨어 개발자들은 서로가 이점을 만들어 주어야 한다. 나는 언젠가 사유 소프트웨어가 함께 이용할 수 없는 대규모 GPL 라이브러리를 우리가 갖게 되길 희망한다. 그것은 새로운 자유 소프트웨어 작업에 유용한 모듈을 제공하고, 더 나은 자유 소프트웨어 개발에 중요한 이점을 더할 것이다.

## 가려운 곳을 긁는다?

에릭 레이먼드 Eric E. Raymond는 ‘소프트웨어에 있어 모든 좋은 성과는 개발자 자신의 가려운 곳을 긁는 것으로부터 시작된다’<sup>41</sup>라고 말한다.

때때로 이는 타당한 말일 것이다. 그러나 GNU 소프트웨어의 많은 핵심 부분은 완전한 자유 운영체제를 만들기 위한 목적으로 개발된 것이다. 가려울 때의 충동이 아닌 비전과 계획에 따른 것이다.

예를 들면 GNU C 라이브러리는 유닉스 유사 운영체제에 C 라이브러리가 필요했기 때문에 개발된 것이고, 배시와 GNU tar<sup>Tape ARchiver</sup>도 마찬가지다. 우리가 개발한 프로그램인 GNU C 컴파일러와 GNU 이맥스, GDB<sup>Gnu DeBugger</sup>, GNU Make 모두 같은 경우다.

어떤 GNU 프로그램은 자유에 대한 위협에 대처하기 위해 개발됐다. Gzip<sup>Gnu ZIP</sup>은 LZW<sup>Lempel-Ziv-Welch</sup> 특허 때문에 공동체에서 사라진 압축 프로그램을 대신하기 위해 개발됐다. 우리는 특정한 자유 라이브러리로 인해 발생한 문제를 해결하기 위해 레스티프<sup>LessTif</sup>를 개발할 사람을 찾았고, 더욱 최근에는 그놈<sup>GNOME</sup>과 하모니<sup>Harmony</sup> 개발에 착수했다.<sup>42</sup> 또한 인기 있는 비자유 암호화 소프트웨어를 대신하기 위해 GnuPG<sup>GNU Privacy Guard</sup>를 개발하는 중인데,<sup>43</sup> 그 이유는 사용자가 사생활과 자유 중 하나만을 선택해야 돼서는 안 되기 때문이다.

물론 이 프로그램을 만든 사람들은 작업에 흥미를 느꼈고, 또한 필요와 관심을 가진 다양한 사람에 의해 많은 기능이 추가되었다. 그러나 이것이 프로그램이 생겨난 이유는 아니다.

## 예기치 않은 개발

처음 GNU 프로젝트를 시작할 때는 모든 프로그램을 완성한 후에 GNU 시스템 전부를 다 함께 내놓으려고 했는데 다음과 같은 이유로 그렇게 되지 못했다.

GNU 시스템의 각 구성 요소는 유닉스 시스템에서 먼저 구현되기 때문에 완전한 GNU 시스템이 존재하기 오래 전부터 유닉스에서 동작할 수 있었다. 이러한 프로그램 중 일부는 인기를 얻었고 사용자가 프로그램을 확장하거나, 호환되지 않던 다

양한 종류의 유닉스로 이식되기 시작했다. 간혹 유닉스 이외의 운영체제로 이식되기도 했다.

이런 일들은 프로그램을 훨씬 더 강력하게 만들었고, 더 많은 기금과 기여자를 GNU 프로젝트로 끌어들였다. 그러나 최소의 동작 가능한 시스템을 완성하는 것 또한 몇 년 뒤로 지연됐다. 왜냐하면 GNU 개발자들이 다른 구성 요소를 만들기보다 이런 프로그램의 이식이나 기능 추가 등에 더 많은 시간을 쏟아야 했기 때문이다.

## GNU 허드

GNU 시스템은 1990년에 거의 완성되었다. 유일하게 빠진 핵심 요소가 커널이었는데, 우리는 마크<sup>44</sup>를 기반으로 커널을 구현하기로 했다. 마크는 카네기 멜론 대학교 Carnegie Mellon University에서 개발된 후 유타 대학 University of Utah으로 이어진 마이크로커널이다.<sup>45</sup> GNU 허드 HURD<sup>46</sup>는 마크와 그 위에서 실행되는 프로세스 서버(데몬)들의 집합으로, 유닉스 커널의 다양한 작업을 수행한다. 마크는 자유 소프트웨어로 공개될 예정이었기 때문에 그때를 기다리며 개발이 지연되었다.

마크를 이용하기로 한 이유 중 하나는 소스 수준의 디버거 없이 커널 프로그램을 디버깅해야 하는 가장 어려워 보이는 작업을 피하기 위해서였다. 마크에는 이 작업이 이미 끝나 있기 때문에 우리는 허드 서버 프로그램을 일반 프로그램처럼 GDB로 디버깅할 수 있으리라 기대했다. 그러나 이것이 가능하게 되기까지는 긴 시간이 걸렸다. 또한 메시지를 서로 주고받는 멀티스레드 서버들은 디버깅하기 매우 힘든 것으로 드러났다. 결국 허드를 확실히 동작하게 하는 데는 많은 시간이 필요했다.

## 알릭스

GNU 커널 이름이 처음부터 허드였던 것은 아니다. 원래 이름은 당시 내 애인의 이름을 딴 알릭스 Alix였다. 유닉스 시스템 관리자였던 그녀는, 알릭스라는 이름이 일

반적으로 사용되던 유닉스 시스템 이름 형식과 잘 어울린다는 얘기를 했는데, 친구들에게 농담 삼아 “누군가 내 이름을 커널 이름으로 써야 할 거야!”라고 말하기도 했다.

나는 아무 말도 하지 않았지만 커널 이름을 알리스로 해서 그녀를 놀라게 해주리라 마음먹었다. 그러나 내 생각은 실현되지 못했다. 커널의 주요 개발자였던 마이클 부슈넬 Michael Bushnell<sup>47</sup>이 허드라는 이름을 더 선호했기 때문이다. 알리스는 허드 서버에 메시지를 전송하는 방법으로 시스템 콜을 제어하는 커널의 한 부분 이름으로 바뀌었다.

그 뒤에 나는 알리스와 헤어졌고 그녀는 이름을 바꿨다. 이와 무관하게 허드는 C 라이브러리가 서버로 메시지를 직접 보낼 수 있게 설계가 수정되었는데, 이 때문에 커널 한 부분에 붙인 알리스란 이름도 사라지게 됐다.

그러나 이런 일이 일어나기 전에 알리스의 친구 중 한 명이 우연히 허드 소스 코드에 있던 알리스란 이름을 보고 그 사실을 전해주었다. 결국 그녀는 자신의 이름을 딴 커널의 존재를 알게 되었던 것이다.

## 리눅스와 GNU/리눅스

GNU 허드는 최종 사용자용으로 적합하지 않으며, 그렇게 될 수 있을지 아직 알 수 없다. 마이크로커널의 권한 capability 중심 설계<sup>48</sup>는 그 유연성 때문에 직접적으로 발생하는 문제가 있는데, 문제에 대한 해결 방법이 존재하는 지가 명확하지 않다.

다행히도 또 다른 커널이 있었다. 리눅스 토르발스 Linus B. Torvalds는 1991년에 유닉스 호환 커널을 개발하며 이름을 리눅스 Linux<sup>49</sup>라고 지었다. 그리고 1992년에는 이를 자유 소프트웨어로 만들었다. 우리는 완전하지 않은 GNU 시스템에 리눅스를 결합해 완전한 자유 운영체제를 만들 수 있었다(이들을 결합하는 것은 물론 그 자체로 상당한 양의 작업이었다).

실제로 작동하는 현재의 GNU 시스템은 리눅스가 있었기에 가능했다. 우리는 이 운영체제를 GNU와 리눅스 커널의 결합으로 표현하기 위해 GNU/리눅스<sup>50</sup>라고 부른다.

## 미래의 도전들

우리는 광범위한 자유 소프트웨어를 개발함으로써 역량을 증명해 왔다. 그러나 이 사실이 우리가 무소불위라는 것을 의미하진 않는다. 여러 도전이 자유 소프트웨어의 앞날을 불확실하게 만들고 있다. 이를 넘어서려면 변함없는 노력과 인내, 때로는 수년간 지속될 시간이 요구될 것이다. 또한 자유의 가치를 소중히 하고 이것을 지켜야 할 때 자기 생각을 분명히 표출하는 결단도 필요할 것이다.

다음 다섯 개 소단원에서 이런 도전에 대해 논해 본다.

### 비밀 하드웨어

하드웨어 제조업체가 기술명세 specifications를 비밀로 하는 경향이 점점 더 뚜렷해지고 있다. 이는 리눅스와 XFree86에서 사용할 새로운 하드웨어의 자유 드라이버 개발을 힘들게 한다. 우리는 지금 완전한 자유 운영체제를 갖고 있지만 미래의 컴퓨터를 지원할 수 없다면, 계속해서 자유 운영체제를 가질 수는 없을 것이다.

이 문제에는 개발자와 사용자 차원의 두 가지 대처 방법이 있다. 개발자의 대처 방법은 하드웨어를 지원할 방법을 찾기 위해 리버스 엔지니어링 reverse engineering을 이용하는 것이고, 사용자의 대처 방법은 하드웨어를 구매할 때 자유 소프트웨어가 지원하는 하드웨어를 택하는 것이다. 이 대처 방법들을 선택하는 우리의 숫자가 많아 질수록 제조업체의 기술명세 비밀 유지 정책은 자멸을 불러오는 것이 될 것이다.

리버스 엔지니어링은 매우 큰 작업이다. 그렇다면 이 일을 수행할, 충분히 결단력 있는 개발자를 찾을 수 있을까? 자유 소프트웨어는 원칙의 문제며, 비자유 드라이버

를 수용할 수 없다는 분위기를 강하게 확립한다면 가능하다고 생각한다. 우리 중 많은 사람이 여분의 돈을, 나아가 조금의 여유 시간을 투자한다면 자유 드라이버를 사용할 수 있지 않을까? 자유를 갖겠다는 결단이 확산되면 그렇게 될 수 있을 것이다.

이 문제는 BIOS로 확대된다. 우리에겐 자유 BIOS인 코어부트 coreboot가 있지만, 문제는 코어부트가 하드웨어를 지원할 수 있게 기술명세를 얻는 것이다.

## 비자유 라이브러리

자유 운영체제에서 작동하는 비자유 라이브러리는 자유 소프트웨어 개발자에게 덫처럼 작용한다. 라이브러리의 매력적인 기능이 미끼다. 이런 라이브러리를 사용한 프로그램은 자유 운영체제에 유효하게 포함될 수 없는 덫에 빠진다(정확하게 말하면, 프로그램은 포함될 수 있지만 라이브러리를 포함시킬 수 없기 때문에 프로그램이 실행되지 않는다). 더욱 좋지 않은 경우는 비자유 라이브러리를 사용한 프로그램이 인기를 얻었을 때다. 이것은 의심 없는 다른 개발자를 같은 함정에 빠뜨릴 수 있다.

이런 경우의 첫 번째 실례는 1980년대의 모티프 Motif 툴킷이다. 당시에는 아직 자유 운영체제가 존재하지 않았지만 모티프가 어떤 결과를 초래하리라는 것은 이미 분명했다. GNU 프로젝트는 두 가지 방법으로 여기에 대응했다. 첫째는 자유 소프트웨어 프로젝트들이 모티프뿐 아니라 자유 X 툴킷 위젯도 지원하도록 요청한 것이고, 두 번째는 모티프를 대신할 자유 소프트웨어를 만들 사람을 찾는 것이었다. 이 일에는 많은 시간이 소요됐는데, ‘헝그리 프로그래머스’<sup>51</sup>가 개발한 레스티프 LessTif는 1997년에 와서야 대부분의 모티프 애플리케이션을 지원하기 충분한 대체물이 되었다.

1996년과 1998년 사이에는 또 다른 비자유 GUI 툴킷 라이브러리 Qt가 나타났다. Qt는 실질적으로 자유 소프트웨어를 모아놓은 환경인 KDE<sup>K Desktop Environment</sup> 데스크톱에서 사용된다.

우리는 Qt 라이브러리를 사용할 수 없기 때문에, 자유 GNU/리눅스 시스템에서는 KDE를 이용할 수 없다. 그러나 GNU/리눅스 시스템을 판매하는 상업 배포업체 중 자유 소프트웨어를 고수하는데 엄격하지 않은 몇몇은 KDE를 포함시키기도 했는데, 이는 시스템 능력은 향상시킨 것이지만 자유는 저하시킨 것이다. KDE 모임은 더 많은 개발자가 Qt를 사용하도록 적극적으로 권장했고, 리눅스를 새로 접한 수많은 사람은 이러한 문제를 미처 알지 못한 채 KDE를 사용하게 되었다. 상황이 암울해 보였다.

자유 소프트웨어 공동체는 그놈과 하모니로 이 문제에 대응했다. 그놈<sup>52</sup>은 미겔 데 이카사Miguel de Icaza<sup>53</sup>에 의해 1997년부터 시작되어 레드햇의 지원으로 개발된 GNU 데스크톱 프로젝트다. 그놈은 유사한 데스크톱 편의를 제공하지만, 자유 소프트웨어만을 사용해 만들어졌다. 또한 C++ 뿐 아니라 다양한 언어를 지원하는 등의 기술적 장점도 있다. 그러나 그놈의 주된 목적은, 어떠한 비자유 소프트웨어의 사용도 요구되지 않는 자유였다. 하모니는 KDE 소프트웨어를 Qt 없이 실행할 수 있는 대체 가능한 호환 라이브러리다.

1998년 11월, Qt 개발자들은 Qt 이용허락을 변경할 것이라고 발표했다. 만약 그렇게 된다면 Qt는 자유 소프트웨어가 된다. 확신할 순 없지만, 나는 이것이 Qt가 비자유 소프트웨어일 때 취했던 태도에 대한 공동체의 단호한 대응이 부분적으로 영향을 미쳤다고 생각한다. 그러나 Qt의 새로운 이용허락은 불충분하므로 아직은 Qt를 사용하지 않는 것이 바람직하다(새소식: 2000년 9월에 Qt는 GNU GPL로 출시되었다. 따라서 이 문제는 이제 근본적으로 해결되었다).

또 다른 비자유 라이브러리의 유혹에 우리는 어떻게 대응해야 할까? 덧에서 벗어나 있어야 할 필요성을 공동체 모두가 인식하게 될까? 아니면 우리 중 많은 사람이 편리함 때문에 자유를 포기하고 같은 문제를 만들어 낼까? 우리의 미래는 우리의 철학에 달려 있다.

## 소프트웨어 특허

우리가 당면한 가장 큰 위협은 소프트웨어 특허로부터 온다. 알고리즘과 새로운 기술에 대한 특허는 최대 20년까지 자유 소프트웨어를 배제시킨다. LZW 압축 알고리즘 특허는 1983년에 출원됐는데, 우리는 제대로 된 GIF<sup>Graphics Interchange Format</sup> 압축 기능이 있는 자유 소프트웨어를 아직 내놓지 못하고 있다. 1998년에는 특허 소송의 위협 때문에 MP3 오디오 압축 자유 프로그램을 배포판에서 제외하기도 했다(새소식: 이 글을 쓴 때는 1998년이다. LZW 특허는 2009년에 만료되었다<sup>54</sup>).

특허에 대처할 방법은 특허가 유효하지 않다는 증거를 찾거나, 같은 기능을 하는 대체 방법을 찾는 것인데 모든 경우에 적용될 수 있는 것은 아니다. 또한 두 가지 방법이 모두 실패하면, 사용자가 원하는 특정 기능이 모든 자유 소프트웨어에서 제외될 수 있다. 만약 이런 일이 벌어진다면, 우리는 어떻게 해야 할까?

자유 소프트웨어가 가진 자유의 가치를 중요하게 생각하는 사람은 어떤 상황에서도 자유 소프트웨어를 버리지 않을 것이다. 우리는 특허와 관련된 기능이 없더라도 나름대로 원하는 작업을 해 나갈 수 있을 것이다. 그러나 기술적 우위를 기대하며 자유 소프트웨어에 가치를 부여했던 사람은 기술적 우위가 특허 때문에 뒤쳐진 후에는 자유 소프트웨어가 실패했다고 말하기 쉬울 것이다. 따라서 ‘시장’ 개발 모델<sup>55</sup>이 가진 실용적인 유효성과 몇몇 자유 소프트웨어가 가진 위력과 신뢰성에 대해 말하는 것이 유용한 동안에는 단지 기술적 우위에 대해서만 이야기해서는 안 된다. 우리는 자유와 원칙에 대해서도 함께 이야기해야만 한다.

## 자유 문서

자유 운영체제에 제일 부족한 것은 소프트웨어가 아니라 함께 포함해야 할 양질의 문서다. 문서 자료는 모든 소프트웨어에 있어 필수적인 부분이다. 중요한 자유 소프트웨어 패키지에 좋은 자유 매뉴얼이 없다면 이는 심각한 빈틈이라고 할 수 있는데, 우리에겐 지금 그런 빈틈이 많이 있다.

상업적 판매를 포함해서 문서의 온라인 및 인쇄물 재배포가 허용되어 프로그램의 모든 복제물에 매뉴얼이 함께 제공되어야 한다. 이때 자유 소프트웨어와 마찬가지로 자유 문서에는 가격이 아닌 자유가 중요 관건이 된다. 자유 매뉴얼의 판단 기준은 자유 소프트웨어와 거의 같다. 즉, 모든 사용자에게 특정한 자유를 허용하는지 여부이다.

개작에 대한 허용 또한 필수적이다. 그러나 나는 일반적으로 모든 종류의 글과 책을 개작할 수 있도록 허용해야 한다고 생각하지 않는다. 예를 들어 여러분이 읽는 바로 이 글처럼 우리의 생각이나 활동을 담은 글을 개작할 수 있도록 허용할 필요는 없을 것이다.<sup>56</sup>

그러나 자유 소프트웨어에 대한 문서 개작의 자유가 중요한 데는 특별한 이유가 있다. 소프트웨어를 개작할 권리를 행사해 기능을 수정하거나 추가한 경우, 양심적인 사람이라면 매뉴얼 또한 수정할 것이다. 그러면 개작한 프로그램에 맞는 정확한 문서를 함께 제공할 수 있다. 개발자로 하여금 양심적일 수 있게 그리고 작업을 잘 끝낼 수 있게 허용하지 않는 비자유 매뉴얼은 우리 공동체의 필요를 채우지 못한다.

개작이 어떻게 이루어져야 하는가 등에 대한 제한은 문제가 되지 않는다. 예를 들면 원저자의 저작권 고지나 배포 규정, 저자 명단 등을 그대로 유지하라는 요구는 상관없다. 또한 개작한 문서에 문서가 개작되었다는 사실을 포함시켜야 한다는 것도 무방하다. 심지어 기술적인 내용을 담은 것이 아닌 한, 삭제하거나 수정하지 않아도 되는 부분은 모두 그대로 남겨두라는 제한도 괜찮다. 이러한 종류의 제한은 문제가 되지 않는다. 왜냐하면 양심적인 개발자가 개작한 프로그램에 맞게 매뉴얼을 고치는 작업을 방해하지 않기 때문이다. 바꿔 말하면 이런 것들은 자유 소프트웨어 공동체가 매뉴얼을 충분히 활용하는 것을 막지 않는다.

그러나 기술적인 내용과 관련된 부분은 모두 수정할 수 있어야 하며 개작된 문서는 일반적인 모든 방법과 매체를 통해 배포될 수 있어야 한다. 그렇지 않으면 이러한

제한이 공동체의 장애가 되고, 매뉴얼은 자유롭지 못하게 되어 우리는 또 다른 매뉴얼이 필요하게 된다.

그렇다면 자유 소프트웨어 개발자들이 모든 범위의 자유 매뉴얼을 만들 필요성을 인식하고 이를 실천할 수 있을까? 반복하지만 우리의 미래는 다름 아닌 우리 자신의 철학에 달려 있다.

### 우리는 자유에 대해 이야기해야만 한다

지금은 대략 천만 명 정도의 사람이 ‘데비안 GNU/리눅스’나 ‘레드햇 리눅스’와 같은 GNU/리눅스 시스템을 사용할 것으로 추산한다.<sup>57</sup> 자유 소프트웨어는 순전히 실용적인 이유로 리눅스에 모여든 사람들이 쫓는 실용적인 이점을 계속 발전시켜 왔다.

이것의 긍정적인 결과는 분명하다. 자유 소프트웨어 개발에 대한 더 많은 관심을 불러오고, 자유 소프트웨어 사업에 더 많은 고객이 모일 것이다. 그리고 자유 소프트웨어 대신 상업용 자유 소프트웨어 제품을 개발하도록 기업을 북돋울 수 있는 더 큰 역량도 생길 것이다.

그러나 자유 소프트웨어의 바탕이 된 철학에 대한 인식보다 소프트웨어 자체에 대한 관심이 훨씬 빨리 증가하는 것은 문제다. 따라서 앞에서 설명한 위협과 도전에 맞서는 우리의 능력은 자유를 위해 굳건히 서겠다는 의지에 달려있다. 우리 공동체가 이러한 의지를 확고히 하기 위해서는 새로운 사용자가 공동체로 들어올 때 그런 생각들을 전해주어야 한다.

그러나 실제로는 그렇게 하지 못하고 있다. 새로운 사용자에게 공동체의 윤리를 심어주는 것보다 우선 공동체 안으로 끌어들이려는 노력이 너무 앞섰던 것이다. 우리는 두 가지 모두를 해야 한다. 그리고 두 가지 노력이 균형을 이루도록 힘써야 한다.

## 오픈 소스

새로운 사용자에게 자유에 대해 이야기하는 것이 1998년에는 더욱 어려워졌다. 왜냐하면 공동체 일부가 ‘자유 소프트웨어’ 대신 ‘오픈 소스 소프트웨어’라는 용어를 사용하기로 했기 때문이다.

이 용어를 선호하는 사람들의 의도는 자유를 의미하는 단어 free가 가진 무료(無料)와 자유(自由)라는 두 가지 뜻의 혼동을 피하려는 것인데, 이는 타당한 것이라 할 수 있다. 그러나 또 다른 사람들은 자유 소프트웨어 운동과 GNU 프로젝트에 동기를 부여한 정신을 배제하고, 대신 기업 대표와 고객의 호감을 사려는 목적을 갖고 있다. 그들 중 다수는 자유와 공동체 그리고 원칙보다 이윤을 더 우선시하는 생각을 갖고 있다. 그래서 ‘오픈 소스’라는 표현은 고품질의 강력한 소프트웨어를 만들 수 있는 잠재력을 맛추지만 자유와 공동체 그리고 원칙을 퇴색시킨다.

‘리눅스 ○○○’처럼 리눅스라는 단어를 이름에 사용한 잡지가 좋은 예다. 이런 잡지는 GNU/리눅스에서 작동하는 자유 소프트웨어 광고로 가득 차 있다. 모티프나 Qt와 같은 라이브러리가 다시 나타났을 때, 이런 잡지는 개발자에게 그 위험성을 경고해줄까 아니면 그 제품의 광고를 실을까?

기업의 지원은 여러 형태로 공동체에 도움을 줄 수 있고, 그 밖의 다른 것들도 모두 다 유용하다. 그러나 자유와 원칙에 대해 밀하지 않는 방법으로 얻은 기업의 지원은 화가 될 수 있다. 이는 앞에서 언급한 공동체로 사용자를 끌어들이려는 것과 의식 교육 사이의 불균형을 더욱 악화시킨다.

‘자유 소프트웨어’와 ‘오픈 소스’는 대체로 같은 범주의 소프트웨어를 가리키는 말이다. 차이가 있다면 소프트웨어 자체와 가치 중에서 무엇을 더 중요하게 생각하느냐는 것이다. GNU 프로젝트는 단지 기술이 아닌 ‘자유’의 이념을 표현하기 위해 ‘자유 소프트웨어’라는 용어를 계속 사용하고 있다. 이것은 중요한 것이다.

## 한번 해보자!

'해본다는 건 없다'는 요다의 금언<sup>58</sup>은 멋지게 들리지만 내게는 효과가 없는 말이었다. 해낼 수 있을까라는 불안과, 목표를 이루기에 충분할까라는 불확실 속에서 나는 대부분의 일을 해 왔다. 그러나 내 도시와 적 사이엔 오직 나밖에 없었기 때문에, 나는 어떻게든 노력했으며 때로는 자신이 놀랄 정도로 성공하기도 했다.

때로는 나는 실패해서 내 도시 중 몇몇은 함락되었다. 그러나 또 다른 도시가 위협 받는 것을 보며 다시 전투를 준비했다. 시간이 흐르면서 나는 다른 해커들의 동참을 요청하며, 위협을 감지하는 방법과 위협 속에 진지를 구축하는 방법을 알게 되었다.

그러나 나는 이제 대부분 혼자가 아니다. 전선을 지키려고 사투하는 해커 부대를 보면, 나는 큰 기쁨과 위안을 느낀다. 그리고 나는 실감한다. 아마도 이 도시는 당분간 진재하리라. 그러나 위험은 매년 커지고, 이제 마이크로소프트는 우리 공동체를 노골적인 공격 목표로 삼고 있다. 미래의 자유는 당연히 가질 수 있는 것이다. 그렇게 생각하지 마라! 여러분의 자유를 지키고 싶다면, 그것을 지킬 준비를 해야만 한다.

---

01 역자주\_MIT 인공지능연구소(Massachusetts Institute of Technology, Artificial Intelligence Laboratory)는 컴퓨터과학&인공지능연구소(<http://www.csail.mit.edu/>)로 이름이 바뀌며 현재까지 이어지고 있다.

02 역자주\_해커(hacker)라는 단어가 시스템의 보안을 허물고 해악을 끼치는 범법자의 의미로 흔히 사용되는 이유는 대중매체의 영향 때문이라고 할 수 있다. 본래 해커란 단어는 '프로그램을 만들기 좋아하고 능숙하게 다룰 수 있는, 즉 프로그램 자체를 즐기는 사람'이란 뜻이다. 해악을 끼치는 사람은 크래커(cracker)라고 표현하는 것이 옳다.

- 03 역자주\_PDP(Programmed Data Processor) 시리즈의 10번째 모델. PDP-1부터 PDP-16까지 생산되었다. MIT 인근에 있었던 DEC(Digital Equipment Corporation)는 1998년에 컴팩(Compaq: COMPAtibility And Quality)으로 인수·합병되었으며, 컴팩은 2002년에 HP(Hewlett-Packard)로 인수·합병되었다. DEC는 ‘디·이·시’나 ‘데크’ 또는 ‘디지털’이라고 호칭한다. 본문에 나오는 영어 약자 중 별도의 한글 표기가 없는 경우는 MIT(엠·아이·티)나 PDP(피·디·피)의 경우처럼 각각의 철자를 모두 발음하면 된다. <역자주 13>에서 설명할 약어(두문자어와 축약어)와 관련해서, 이 번역문은 한 단어처럼 발음되는 약어인 두문자어는 ‘유닉스’나 ‘리눅스’, ‘컴팩’처럼 한글로 표기하고 이에 대한 영문 표기 형태는 유럽연합 집행위원회 번역총국 영어 스타일 가이드(<http://www.fjfhs.eu/esg/>)를 기준으로 삼았다. 단, 해당 용어를 만든 주체가 사용하는 관용 표기가 있을 경우에는 그 형태를 따랐다. 또한 번역 형식과 관련해서 본문에 언급된 사람의 경우, 사망한 경우에 한해 생몰연대를 삽입했기 때문에 생몰연대가 병기되지 않은 사람은 혼존 인물로 생각하면 된다.
- 04 역자주\_스핀오프(spin-off)란 대학이나 기업 연구실의 연구원들이 연구 결과를 갖고 독립해 설립하거나 깊이 관여하는 회사를 말한다.
- 05 역자주\_이 책의 초판은 1984년에 출판되었으며 최신판은 같은 제목으로 나온 『Hackers: heroes of the computer revolution, O'Reilly, 2010』이다. 한국에는 『해커, 그 광기와 비밀의 기록, 김동광 옮김, 사민서각, 1996년』, 『해커스, 세상을 바꾼 천재들, 박재호, 이해영 함께 옮김, 한빛미디어, 2013년』 두 권의 다른 번역판이 출간되었다. 1996년 사민서각 번역판의 에필로그 ‘진정한 해커의 종말, 케임브리지 1983년’ 부분에 리처드 스톨먼과 MIT 인공지능연구소를 배경으로 일어난 자세한 뒷이야기가 수록되어 있다. 에른스트 블로흐(Ernst Bloch, 1885~1977)는 『희망의 원리(1), 박설호 옮김, 솔, 1995년』에서 다음과 같이 말한다. ‘모든 유토피아는, 그것이 의학적이든, 사회적이든, 기술적이든 간에 편집광증 환자들에 의해 그려진 캐리커처라고 해도 과언이 아니리라. 왜냐하면 환상에 사로잡히고 비현실적인 태도를 취하던 수백 명의 미친 사람들은 한결같이 주어진 현실을 변화시키려던 선구자였기 때문이다(175페이지).’
- 06 역자주\_<역자주 8> 참고.
- 07 역자주\_VAX(Virtual Address Extension)와 68020은 DEC와 모토로라(Motolora)가 만든 32비트 시스템이다.
- 08 역자주\_2000년 종이책에는 proprietary software를 독점 소프트웨어로 번역했으나, 2013년 전자책에는 공유(共有)의 반대면서 독점보다 넓은 의미를 가진 사유(私有) 소프트웨어로 옮긴다. 이 번역문에서 사용한 소프트웨어 저작권 관련 용어는 다음과 같다.
- 오픈 소스 소프트웨어(open source software)

- 폐쇄 소스 소프트웨어(closed source software)
  - 자유 소프트웨어(free software)
  - 비자유 소프트웨어(nonfree software)
  - 공중영역 소프트웨어(public domain)
  - 사유 소프트웨어(proprietary software)
  - 이용허락(license)
  - 지식재산권(intellectual property rights)
- 09 역자주\_저작권 침해행위와 침해자를 뜻하는 영어 단어 piracy와 pirate는 부정적인 의미를 강조하기 위해 본래 해적행위를 뜻하는 단어에 의미를 추가한 것이다. 한국어에서는 단지 저작권침해라고 표현되지만, 영어에서는 해적행위라는 뜻을 함께 전달한다.
- 10 역자주\_벤더(vendor)나 소프트웨어 발행사(software publisher) 등은 미국과 한국의 사업 형태 차이 때문에 한국어로 번역할 경우 때에 따라 혼동이 생길 수 있다. 소프트웨어 발행사는 이 글 전체에서 두 번 등장하는데, 모두 이 절에서 사용된다. 종이책을 내는 출판사(publisher)를 생각하면 소프트웨어 발행사를 쉽게 이해할 수 있다. 출판사는 저자로부터 원고를 받아 편집 및 인쇄를 마친 뒤에 자체 규모가 있다면 책을 직접 유통하지만 그렇지 않을 경우 유통회사와 계약해 책을 유통 및 판매한다. 경우에 따라 직영 서점이나 인터넷 서점을 운영할 수도 있다. 판매를 통해 얻은 이익은 저자에게 자급되는 인세 등을 빼면 출판사 몫이 된다. 또한 책을 쓴 저자가 출판사로 찾아와 출판을 의뢰하는 경우도 있지만 출판사가 주도적으로 시장성 있는 책을 기획하고 저자를 구해 집필하게 하거나 외국의 책을 번역·출간하기도 한다. 이러한 역할 중 일부 또는 모두를 소프트웨어 분야에서 하는 것이 미국의 소프트웨어 발행사다. 이 절에서 설명되는 내용에 대해 소프트웨어 발행사를 출판사로 바꾸고, 전자책 1인 출판이 가능한 시대 상황을 함께 고려해 보면 또 다른 이해가 생길 수 있다.
- 11 역자주\_이때의 상황은 리처드 스톤먼의 전기 『Free as in Freedom: Richard Stallman's Crusade for Free Software, Sam Williams, O'Reilly, 2002』 제7장(<http://oreilly.com/openbook/freedom/ch07.html>)에서 자세히 참고할 수 있다.
- 12 역자주\_일반적으로 '그누'로 발음하는 GNU는 GNU 프로젝트와 GNU 시스템을 지칭하는 데 공통으로 사용한다. 단, GNU 시스템은 GNU 프로젝트를 통해 구현한 유닉스와 호환되는 완전한 운영체제 전체를 말하고, GNU 프로젝트는 이러한 시스템을 구현하기 위해 진행되는 프로젝트 자체를 의미한다. GNU의 발음은 <http://www.gnu.org/pronunciation/pronunciation.ko.html>에서 들을 수 있다. 사파리와 IE 브라우저는 ogg(오그) 포맷을 지원하지 않기 때문에 <http://www.gnu.org/audio/gnu-pronunciation.ogg>에서 파일을 직접 다운로드해서 들을 수 있다. GNU는 특허와 독점 파일 포맷을 거부하기 때문에 mp3(MPeg audio layer-3)가 아닌 ogg 포맷을 사용한다.

- 13 역자주\_문장을 구성하는 단어의 첫 문자를 모아 만든 약어를 두문자어(acronym)라 한다. GNU는 'GNU's Not Unix', 즉 'GNU는 유닉스가 아니다'라는 뜻이 되도록 단어를 처음부터 의도적으로 조합해 만든 것이다. 예를 들면, EINE(아이네)라는 이름의 텍스트 편집 프로그램은 이미 존재하는 프로그램인 Emacs와 다른 것이라는 의미로 '○○○ Is Not Emacs'라는 문구를 먼저 생각한 뒤에 첫 문자를 E로 결정한 후 문장의 각 단어 첫 문자를 따와 EINE라는 이름이 되도록 만든 것이며, Cygnus(시그너스)는 'Cygnus: Your GNU Support'라는 문구를 이용해 만든 것이다. 이러한 조어 형태를 재귀적 두문자어(recursive acronym)라 한다. GNU는 유닉스와 호환되게 만든 운영체제지만 유닉스와는 다른 운영체제라는 의미를 담은 재귀적 두문자어다. Free Software Foundation을 줄여 상황에 따라 FSF로도 표기하는데, 이런 형태를 축약어(initialism)라고 한다. 두문자어와 축약어는 모두 약어(abbreviation)이며, 외면상 같지만 컴퓨터 분야에서 의도적으로 만들어지는 두문자어를 구분하기 위해 여기서는 편의상 따로 구분한다. 두문자어와 축약어는 대소문자 표기와 발음에 차이가 있다(〈역자주 3〉 참고). 예를 들면, Emacs(이맥스)는 두문자어고 FSF(에프·에스·에프)는 축약어다.
- 14 역자주\_뒷부분에서 자세히 설명한다. <[역자주 45](#)> 참고.
- 15 역자주\_Multics는 'MULTIplexed Information and Computing Service'의 두문자어다. 첫 단어의 접두사 multi가 의미하듯이 다중사용자와 멀티태스킹 기능을 이용해 일을 효율적으로 수행할 수 있다는 뜻이지만, 구현 기술상의 문제로 기대만큼 좋은 운영체제가 되지 못했다. 그러나 멀티태스킹과 같은 새로운 개념은 유닉스 개발에 직접 영향을 주었다. 유닉스는 멀티스 프로젝트에 참여했던 켄 톰프슨(Ken L. Thompson)이 개발했으며 DEC의 PDP-7에 처음 탑재되었다. 유닉스라는 이름은 멀티스의 원래 의미에 대한 짓궂은 반어적 의미로 multi 대신 uni를 사용해 Unix(UNiplexed Information and Computing Service), 즉 한 가지 일이라도 확실히 수행하는 운영체제라는 의미를 담은 것이다. 이 이름은 C 언어 창시자 데니스 리치(Dennis M. Ritchie, 1941~2011)의 고전『The C Programming Language』의 공저자 브라이언 커니핸(Brian W. Kernighan)이 만들었다. 이들은 모두 벨 연구소(AT&T Bell Laboratories)에서 근무한 동료들이다. VMS(Virtual Memory System)는 DEC가 만들어 VAX 컴퓨터에 탑재한 운영체제다.
- 16 원주\_무신론자인 나는 어떤 종교 지도자도 따르지 않지만, 때때로 그들이 남긴 금언에 내가 탄복하고 있다 는 사실을 깨닫곤 한다.(역자주\_힐렐(Hillel, B.C.60~A.D.20)은 예수 탄생 전에 실존한 유대교 합비다. 이 금언은 탈무드 토라의 '위대한 세 명의 합비' 장에서 인용한 것이다.)
- 17 역자주\_영어 단어 free는 한국어의 자유와 무료를 의미하는 두 가지 뜻을 함께 갖기 때문에 영어권에서는 자유 소프트웨어를 무료 소프트웨어로 오해하는 경우가 많다. 이 절은 이에 대한 설명이다. 라틴어에서 유래한 단어 gratis(무료)가 영어에도 쓰이지만, 무료 맥주(free beer)와 언론의 자유(free speech) 두 가지 의미 모두로 가장 많이 사용되는 것은 free다. 한국어에서는 자유와 무료가 비교적 명확히 구분되기 때문에 영

어의 모호함을 피하기 위해서도 free software를 ‘프리 소프트웨어’가 아닌 ‘자유 소프트웨어’로 읊기는 게 바람직하다.

- 18 역자주\_텍(TeX, <http://www.ktug.org/>)은 스탠퍼드 대학교의 도널드 카누스(Donald E. Knuth) 교수가 개발한 조판 프로그램이다. TeX은 기술과 예술을 뜻하는 그리스어  $\tau\epsilon\nu$ 에서 유래한 말로, 원어는 ‘테흐’ 또는 ‘테히’로 발음하지만 일반적으로 텍으로 발음한다(이 단어는 영어 technology의 접두사 tec의 어원이기도 하다). 텍 파일은 수식을 자유롭고 미려하게 편집·조판할 수 있기 때문에 주로 이공계 연구 논문 포맷으로 많이 사용된다.
- 19 역자주\_V는 vrij의 첫 문자이고 자유 대학에서 쓰인 ‘자유’는 종교로부터 자유롭다는 의미다. 리처드 스톤먼이 받은 회신은 네덜란드어가 아닌 영어로 되어 있었기 때문에 free라는 영어 단어가 정확히 어떤 의미로 사용되었는지 자신도 알 수 없다고 한다. 단지 정확하게 말할 수 있는 것은 VUCK가 자유 소프트웨어가 아니었다는 점이다.
- 20 역자주\_타깃 머신(target machine)은 말 그대로 컴파일한 프로그램을 최종적으로 실행할 목표 기계, 즉 시스템을 구분 짓는 아키텍처를 말한다. 통상의 경우 인텔 x86 CPU가 설치된 시스템에서 개발해 컴파일한 프로그램은 동일한 x86 시스템에서만 실행되지만, 타깃 머신을 ARM으로 정해 컴파일하면 x86 CPU와 전혀 구조가 다른 ARM CPU가 탑재된 휴대폰 등에서 실행할 수 있다. 이렇게 특정한 아키텍처에서 실행할 프로그램을 그와 다른 환경에서 개발해 컴파일하는 것을 크로스 컴파일(cross compile) 또는 교차 컴파일이라 하며, 이러한 기능이 있는 VUCK는 크로스 컴파일러의 하나가 된다. 좀 더 자세한 내용은 이 책의 다른 글 ‘시그너스 솔루션즈의 미래’ 중 <역자주 42>에서 참고할 수 있다.
- 21 역자주\_포트란을 비롯한 대부분의 프로그래밍 언어는 프로그램이 사용할 변수를 미리 선언하도록 설계되어 있다. 그러나 파스텔 언어는 임의의 위치에서 변수를 선언할 수 있기 때문에 변수에 대한 정보를 올바르게 유지하려면 먼저 프로그램을 한 번에 모두 읽어 메모리로 옮린 뒤에 처리하는 구조를 가질 수밖에 없다. 이런 방식에서는 소스 코드와 거의 동일한 크기의 메모리와 스택이 필요하며 레지스터와 스택 또한 전혀 효율적으로 사용할 수 없기 때문에 당시의 하드웨어 환경을 고려하면 파스텔 컴파일러는 GNU 시스템에 맞는 것 이 될 수 없었다.
- 22 역자주\_1987년 3월 22일에 최초의 베타판이 공개되었으며, C 언어로 작성한 당시의 컴파일러 이름은 GCC(GNU C Compiler)였다. 그 후 다양한 언어의 프런트 엔드와 여러 부가기능이 추가되어 종합 컴파일러 세트가 되었다. 현재의 명칭은 GCC(GNU Compiler Collection)이며, 10개 이상 언어의 프런트 엔드가 안정적으로 개발되어 있다. 상세한 정보는 GCC 홈페이지(<http://gcc.gnu.org/>)에서 확인할 수 있다.

- 23 역자주\_최초의 Emacs(Editor MACroS 또는 Eight Megabytes And Constantly Swapping)는 리처드 스톰먼이 테코(TECO: Text Editor and COrrector) 편집기를 확장해 만든 1976년 판이다. 그 후 자바 언어의 창시자 제임스 고슬링(James A. Gosling)이 수정한 고슬링 이맥스가 개발됐고, 그 후 리처드 스톰먼에 의해 다시 새롭게 만들어진 것이 이 절에서 설명하는 GNU 이맥스다. 개발 시기로 볼 때, 이맥스는 GNU 프로젝트가 시작되기 전에 개발돼 계속 이어진 것이기 때문에 재귀적 두문자어로 만들어진 이름 Emacs 안에 GNU를 의미하는 G가 포함되어 있지 않다. 본문에서 GNU 프로젝트의 첫 번째 프로그램을 이맥스가 아닌 GCC 컴파일러로 언급한 것도 이러한 이유에서다.
- 24 역자주\_이 이름은 현재까지 유지되고 있다. [ftp://prep.ai.mit.edu/](http://prep.ai.mit.edu/) 또는 <http://prep.ai.mit.edu/>로 접속할 수 있으며, 실제로는 모두 현재의 주된 GNU FTP 주소인 [ftp://ftp.gnu.org/](http://ftp.gnu.org/)와 같다. 같은 이름을 쓰는 <http://www.gnu.org/prep/>에는 GNU와 FSF 조직 관련 문서가 올려져 있다.
- 25 역자주\_당시의 보조 및 이동 기억장치는 CD-ROM과 플로피 디스크 이전 세대인 자기 테이프(magnetic tape, [http://en.wikipedia.org/wiki/DECtape#DECtape\\_II](http://en.wikipedia.org/wiki/DECtape#DECtape_II))다.
- 26 역자주\_당시의 150달러는 1980년대 중반 환율 기준 약 13만 원이며, 통계청이 발표하는 연도별 소비자물가상승률을 고려한 현재 화폐가치는 약 40만 원이다. 당시의 DEC 테이프 가격은 해당 약 22달러(약 1만 9천 원)였다.
- 27 역자주\_이 글에서는 permissive license를 ‘허용적 이용허락’으로 번역한다. 다음 절에서 설명하는 MIT 이용허락은 <http://opensource.org/licenses/mit-license.php>에서 참고할 수 있으며, 자유 소프트웨어 진영에 대응하기 위해 마이크로소프트가 2001년부터 계획한 MS 허용적 이용허락은 <http://www.microsoft.com/korea/resources/sharedsource/licensingbasics/sharedsourcelicenses.mspx#EXB>와 <http://opensource.org/licenses/MS-PL>에서 참고할 수 있다. MS Permissive License는 MS Public License로 명칭이 변경되기도 했다.
- 28 원주\_1984년이나 1985년 즈음에, 매우 상상력 풍부한 친구인 돈 홉킨스(Don Hopkins)가 내게 편지를 보낸 적이 있다. 그가 보낸 편지봉투에는 재미있는 문구가 몇 개 적혀 있었는데, 그 중 ‘카피레프트, 모든 권리가 취소됩니다(copyleft – all rights reversed)’라는 것이 있었다. 나는 ‘카피레프트’라는 단어를 당시 내가 만들던 배포 형태를 뜻하는 말로 사용했다.
- 29 역자주\_‘양도할 수 없는 권리(unalienable rights)’는 미국의 「독립 선언서(Declaration of Independence)」에서 유래한 것으로 창조주가 모든 사람에게 부여한 권리를 말한다. 이 권리는 분리되거나 포기하거나 빼앗아 갈 수 없는 것이다. 독립선언문은 생명과 자유 그리고 행복의 추구를 양도할 수 없는 권리로 규정하고 있는데, 좀더 역사적으로는 존 로크(John Locke, 1632~1704)의 자연법 사상에서 그 유래를 찾을 수 있다. 로크는 생명, 자유, 재산에 대한 권리를 양도할 수 없는 것으로 규정한다.

- 30 역자주\_ 이러한 경우의 쉬운 예로 대중가요를 생각해 볼 수 있다. 한 노래의 작사가와 작곡가가 다를 때, 만약 작사가는 가사를 자유롭게 공개했지만 작곡가는 그렇게 하지 않았다면, 가사는 독립적으로 자유롭게 공유될 수 있지만 가사와 곡이 결합된 노래는 저작권 제한으로 자유롭게 공유될 수 없다.
- 31 역자주\_ 카피레프트는 자유 소프트웨어 이용허락을 위한 법률적 개념이며, 이를 구체적으로 구현한 실제 계약 양식이 GNU GPL이다. 카피레프트에는 GPL, LGPL, FDL 등 많은 종류의 실제 이용허락 양식이 있다.
- 32 원주\_ 문서에는 GNU 자유 문서 이용허락(GNU Free Documentation License, <http://korea.gnu.org/people/chsong/copyleft/fdl-1.2.ko.html>)을 사용한다.
- 33 역자주\_ 자유 소프트웨어 재단은 비영리 법인이며, 비과세를 위한 세법상 구분이 미국 내국세 입법 비영리 단체 종류 구분 503(c)(3)에 따른 자선단체로 되어 있다. 이에 따라 FSF에 대한 기부는 미국 안에서 소득공제 대상이 된다. 그러나 한국 법인세법이 정하는 기부금 단체에는 포함되지 않기 때문에 한국에서 기부한 경우 소득공제가 되지 않는다.
- 34 원주\_ Bash(Bourne Again SHell)라는 이름은 유닉스에서 일반적으로 사용되던 sh(Bourne Shell)에 빗대 만든 재미있는 표현이다. (역자주\_Bash는 다른 셸들과 달리 한 단어처럼 배시라고 발음한다. bourne again은 발음상 born again이 되어 최초의 셸 sh의 재현이라는 의미가 된다. 원주에서 말하는 '재미있는 표현'이란 이런 뜻이다.)
- 35 역자주\_ GNU 프로그램 코딩 표준에 대한 언급이다. 좀 더 자세히 설명하면, 프로그래머가 자료구조나 자료형의 크기를 임의로 제한해 사용하지 않고 동적 할당을 택하게 함으로써 시스템의 안정성과 유연성을 높인 것이다. 7비트 대신 8비트 코드를 도입해 문자 인코딩을 확장한 것 등이다. GNU는 유닉스 사이의 API 표준인 POSIX(Portable Operating System Interface, uniX, 포식)에 대해 '우리는 POSIX를 참고하지만 복종하지는 않는다'는 태도를 보이는데, 이것은 이 절의 내용을 요약한 듯한 표현이다. GNU 코딩 표준은 <http://www.gnu.org/prep/standards/standards.html>를 통해 자세히 참고할 수 있다.
- 36 역자주\_ 메모리 안의 특정 위치는 주소로 구분되기 때문에 컴퓨터가 최대 얼마의 주소, 즉 숫자를 헤아릴 수 있느냐가 사용할 수 있는 메모리 크기를 결정한다. 따라서 16비트와 32비트 메모리 주소를 갖는 컴퓨터는 각각  $2^{16}=64\text{KiB}$ 와  $2^{32}=4\text{GiB}$ 의 메모리를 사용할 수 있다. 그러나 32비트 컴퓨터라고 해도 운영체제가 이를 지원하지 않으면 이론상의 메모리를 모두 사용할 수 없다. GNU가 16비트 컴퓨터를 지원하지 않기로 하면서 cat이나 tail 등과 같은 큰 파일을 제한 없이 입·출력하는 프로그램이 아니라면 프로그래머에게 더 유연한 환경을 제공하게 되었다.
- 37 원주\_ 이 글은 1998년에 쓴 것이다. 2009년부터는 GNU 태스크 리스트를 더 이상 유지하지 않고 있다. 공동체의 자유 소프트웨어 개발이 너무 빠르게 이루어지기 때문에 이제 현황을 파악하는 것조차 힘들다. 대

- 신 최우선 순위 프로젝트 목록(<http://www.fsf.org/campaigns/priority-projects/>)을 만들었는데, 이 것은 개발이 정말 시급한 훨씬 짧은 길이의 목록이다. (역자주\_좀 더 세분된 형태의 프로젝트 참여 요청은 <http://www.gnu.org/software/tasklist/howto-volunteer.html#N0x84a9878N0x9d8d074>에서 참고할 수 있다.)
- 38 원주\_마치 모든 라이브러리에 적용해야 할 듯한 느낌을 없애기 위해서 지금은 Library를 대신 Lesser를 사용해 GNU 약소 일반 공중 이용허락(GNU Lesser General Public License)으로 부른다. 관련 글로 '라이브러리에 LGPL를 사용하지 말아야 하는 이유(<http://www.gnu.org/philosophy/why-not-lgpl-ko.html>)'가 있다.
- 39 역자주\_컴퓨터에서 하는 작업 대부분이 문서 작성인 작가 황마마가 있다고 하자. 만약 황마마가 사유 문서 편집 소프트웨어 오로라를 사용해야만 한다면, (GNU 시스템에는 GNU C 라이브러리만 있으므로, 다시 말해 사유 소프트웨어 제작에 사용된 C 라이브러리가 GNU 시스템에는 없기 때문에) 황마마는 오로라가 실행되지 않는 GNU 시스템을 사용할 현실적인 이유가 없다. 그러나 사유 소프트웨어를 GNU C 라이브러리와 결합할 수 있게 허용하면 오로라 제작업체는 GNU 시스템용 오로라를 출시할 수 있고, 황마마는 기존의 MS 윈도우 환경을 버리고 GNU 시스템용 오로라를 구입해 작업할 수 있다. 황마마가 사유 소프트웨어 오로라 사용자라는 사실은 변치 않았지만, LGPL 전략을 통해 GNU 사용자가 된 것이다. 익숙해진 환경에서 언젠가 오로라를 버리고 이맥스를 사용하게 될지 모른다. 만약 오로라 후속판이나 새로운 프로그램이 GNU 시스템에서만 실행되는 형태로 출시되어 폭넓은 인기를 얻게 될 경우, 여러 이유로 자유 소프트웨어로 전환될 수도 있을 것이다. 이것이 LGPL 전략의 하나다.
- 40 역자주\_명령행 편집 기능(command-line editing)은 셸 프롬프트에 명령어 앞부분을 입력하면, 그 뒤에 이어질 단어 후보가 자동으로 나타나 선택할 수 있는 기능이다. 흔히 웹 브라우저 검색창에 검색어를 입력할 때, 한 글자만 입력해도 해당 문자로 시작되는 검색어 후보가 나타나 키보드나 마우스로 선택할 수 있는 자동 완성 기능이 제공되는데, 이것이 명령행 편집 기능의 작동 형태라 할 수 있다.
- 41 역자주\_에릭 레이먼드의 대표작 「성당과 시장」에 있는 문장이다. <http://korea.gnu.org/people/chsong/cb/cathedral-bazaar/>에서 한국어 번역문을 참고할 수 있다.
- 42 원주\_이 글의 뒷부분 '미래의 도전들' 중 '비자유 라이브러리' 단락에서 자세히 설명한다.
- 43 역자주\_이 글을 쓰여진 1998년에는 GnuPG의 개발이 시작 단계였지만, 1999년 9월 7일 1.0판이 발표된 이후 2013년에는 2.0판이 완성돼 배포되고 있다.
- 44 역자주\_마크 커널의 대표적인 파생물은 GNU 허드와 애플의 맥 OS X다. 맥(Mac)과 구분하기 위해 '마하'나 '마흐'가 아닌 '마크'라 발음한다.

- 45 역자주\_운영체제를 세부적으로 구분할 때 가장 밑단에서 실행되는 핵심 부분을 커널이라고 한다. 커널의 기능을 모듈화해서 이들을 서로 연결하는 구조가 마이크로커널이며, 그렇지 않은 종래의 형태를 모놀리식(monolithic) 커널이라고 한다. 리눅스는 모놀리식 커널이다. 『오픈 소스 Vol. II, 한빛미디어, 2013년』의 부록 A에서 자세한 내용이 설명된다.
- 46 역자주\_GNU의 발음과 철자가 소를 의미하는 일반명사 gnu와 같기 때문에 소머리 그림이 GNU의 상징으로 사용되기도 한다(<http://www.gnu.org/graphics/agnuhead.ko.html>). 허드의 개념은 'GNU 서버(데몬)들의 집합'이기 때문에 이를 'a herd of GNUs'로 쓸 수 있는데, GNU를 일반명사로 해석하면 말 그대로 '소떼'가 된다. 허드라는 이름은 해커 문화의 전통에 따라 이런 다의성과 중첩성을 내포할 수 있게 herd 와 같은 발음이 되도록 일부러 만든 것이다. HURD는 Hird of Unix-Replacing Daemons의 재귀적 두문자이고, 첫 단어 HIRD는 다시 Hurd of Interfaces Representing Depth의 재귀적 두문자이다. 두 개의 두문자어가 서로 물려있는 독특한 구조로 되어 있다.
- 47 원주\_현재 이름은 토머스 부슈넬(Thomas Bushnell)이다.
- 48 역자주\_권한으로 번역한 capability는 마이크로커널 구현에 사용되는 보호된 객체 참조를 뜻하는 용어다. 모든 시스템 서비스와 자원은 마이크로커널이 부여한 접근 권한 표시인 capability에 따라 참조되거나 이용될 수 있다. 자세한 기술 내용은 <http://walfield.org/papers/20070111-walfield-critique-of-the-GNU-Hurd.pdf>를 통해 참고할 수 있다.
- 49 역자주\_Linux(LINUs's uniX)는 리눅스 토르발즈 자신의 유닉스라는 의미로 만든 이름이다. 유닉스 호환 시스템에는 관례적으로 X를 붙이는 경우가 많은데 Linux나 Minix, POSIX, X 윈도 시스템 등이 이 같은 예다.
- 50 역자주\_‘그누 슬래시 리눅스’ 또는 ‘그누 플러스 리눅스’라고 말한다.
- 51 역자주\_헝그리 프로그래머(Hungry Programmers, <http://www.hungry.com/>)는 자유 소프트웨어를 만드는 프로그래머 집단의 이름이다.
- 52 역자주\_GNOME(GNU Network Object Model Environment)의 두문자를 구성하는 단어들은 여러 순서로 배열할 수 있는데, 의미 전달이 가장 명확한 형태는 GNU's Environment Model for Network Objects다.
- 53 역자주\_1999년 미국의 과학 전문지 「MIT Technology Review」 창간 100주년 특집호에서 '다음 세기를 이끌 젊은 과학자 100인'으로 선정되었으며, 같은 해 12월에는 자유 소프트웨어 재단이 선정한 '제2회 자유 소프트웨어 발전을 위한 자유 소프트웨어 재단상'을 수상했다. 자유 소프트웨어 재단은 매년 자유 소프트웨어의 발전에 지대한 공헌을 한 사람을 선정하는데, 첫 번째 상은 프로그래밍 언어 펄(Perl)의 창시자 래리 월(Larry Wall)이 받았다.

- 54 역자주\_ <http://www.gnu.org/philosophy/gif.ko.html>에서 관련 내용을 자세히 참고할 수 있다.
- 55 역자주\_ 성당과 시장 개발 모델은 에릭 레이먼드의 글(<http://korea.gnu.org/people/chsong/cb/cathedral-bazaar/>)에 자세히 설명되어 있다.
- 56 역자주\_ 2000년대 중반 이후부터 이런 목적을 충족하는 보다 간단한 형식인 크리에이티브 커먼즈 이용허락 (Creative Commons License, <http://cckorea.org/xe/?mid=ccl>)이 널리 사용되고 있다. 이 이용허락은 (1) 저작자 표시와 (2) 상업적 이용, (3) 내용 변경, (4) 2차적 저작물에 적용할 이용허락을 선택적으로 조합해 사용할 수 있도록 하면서 아이콘 링크 등을 통해 크리에이티브 커먼즈가 제공하는 설명 페이지를 참고하게 하는 쉬운 방법으로 웹을 통한 창작물의 공유를 장려한다. GNU 홈페이지의 철학 문서들은 다른 사람이 수정하지 못하도록 카피리프트가 아닌 이용허락을 적용했는데, 지금은 CC 이용허락을 병행해서 사용하고 있다.
- 57 역자주\_ 리눅스 카운트 프로젝트(<http://linuxcounter.net/>)를 통해 대략적인 사용자 수를 파악할 수 있는데, 전 세계에는 2013년 8월 현재 약 6천 7백만 명의 리눅스 사용자가 있다. 이 책이 처음 출판된 1999년 이후, 눈에 보이는 숫자로만 여섯 배 이상 증가한 셈이다.
- 58 역자주\_ 영화 스타워즈 에피소드 5에 나오는 제다이 스승 요다(Yoda)의 말(<http://www.youtube.com/watch?v=BQ4yd2W50No>)로 ‘해본다는 것은 의미가 없고 확신을 가지고 실행하는 것만이 포스의 힘을 불러낼 수 있다’는 의미다. 원문의 내용은 다음과 같다. “All right. I'll give it a try. No! Try not. Do... or do not. There is no try(루크 스카이워커: 좋아요, 한번 해볼게요. 요다: 아니야! 한다, 못한다 둘 중 하나야. 한번 해본다는 건 없어).”

## 5 | 시그너스 솔루션즈의 미래: 한 기업가 이야기

마이클 티만 Michael Tiemann

송창훈 역

1989년에 창업한 시그너스 솔루션즈 Cygnus Solutions<sup>01</sup>는 최초의 오픈 소스 회사며, 1998년 8월 포브스 Forbes magazine 조사<sup>02</sup>에 따르면 단연코 현재 가장 큰 오픈 소스 회사다. 시그너스 솔루션즈의 주력 제품 GNUPro 개발자 키트는 임베디드 소프트웨어 도구 시장을 선도하는 컴파일러와 디버거 제품이며 일류 가전제품과 인터넷, 텔레커뮤니케이션, 사무 자동화, 네트워킹, 항공·우주, 자동차는 물론 세계 최고의 마이크로프로세서 회사들이 시그너스 솔루션즈의 고객이다. 캘리포니아 서니베일에 위치한 본사를 비롯해서 미국 애틀랜타와 보스턴, 영국 캐임브리지, 일본 도쿄, 캐나다 토론토에 사무소를 두었고, 오스트레일리아부터 오리건에 이르는 다양한 지역에 원격 근무 직원을 둔 시그너스 솔루션즈는 임베디드 소프트웨어 산업에서 제일 큰 비공개회사<sup>03</sup>이며, 공개회사 두 곳을 넘어 이제 세 번째로 큰 공개회사를 앞지르는 시점에 있다. 또한 1992년부터 65%가 넘는 연평균성장을 CAGR Compound Annual Growth Rate을 유지하며 3년 동안 새너제이 비즈니스 저널 San Jose Business Journal이 선정하는 최고 성장 비공개회사 100위<sup>04</sup> 안에 들었고, 이제 (매출을 기준으로 전 세계 소프트웨어 업체의 순위를 정하는) 소프트웨어 500<sup>05</sup>에 도 포함되어 있다.

나는 이 글에서 우리에게 성공의 청사진을 제공한 오픈 소스 모델과 미래 사업을 위해 이것을 어떻게 바꾸고 발전시켜 나갈 것인지에 대해 설명하려고 한다.

1989년 11월 13일이 되어서야 우리는 마침내 캘리포니아주 법인국<sup>06</sup>으로부터 사업 승인을 알리는 편지를 받았고, 창업 자본 6천 달러(1989년 화폐가치 약 4백만 원, 2013년 화폐가치 약 1천만 원)를 예치한 뒤에 시그너스 서포트 Cygnus Support라는 이름으로 사업을 시작할 수 있었다.<sup>07</sup> 그날은 2년보다 더 오래 거슬러 올라간 때로부터 시작된 계획이 정점에 이른 날이자 지금까지 거의 10년간 지속되고 있는 우리의 여정이 시작된 날이었다.

사업에 대한 희망은 무척이나 순수하게 시작된 것이었다. 한 번은 아버지께서 이런 말씀을 하신 적이 있다. “얘야, 책을 읽으려면 처음부터 끝까지 모두 읽도록 하렴.” 대부분 아버지의 충고를 귀담아듣지 않듯이 나 또한 마음이 내킬 때만 그 말을 따랐는데, 1987년에 하던 일에 짖증을 느끼고 GNU 소프트웨어에 흥미를 갖게 되었을 때는 달랐다. 나는 리처드 스톤먼이 직접 출판한 GNU 이맥스 매뉴얼을 처음부터 끝까지 모두 읽을 결심을 했다(그가 책을 직접 출판한 이유는 책을 구입한 독자가 합법적으로 자유롭게 책을 복제하도록 장려할 출판사가 당시에 없었기 때문이다).<sup>08</sup>

이맥스는 환상적인 프로그램이다. 텍스트 편집기 본연의 기능은 물론이고 사용자 맞춤 설정을 통해 이메일을 처리하고 뉴스그룹에 글을 올리거나 읽을 수 있으며, 편집 화면 안에서 셀을 바로 실행하거나 프로그램을 직접 컴파일하고 디버깅할 수 있다. 심지어 이맥스 프로그램을 구동하는 리스프 LISP, LiSt Processing 인터프리터 자체에 대한 대화식 접근도 가능하다.

창의적인 사용자나 일상적인 것에 짖증을 느낀 해커들은 이맥스에 기발한 기능들을 추가했다. 예를 들면 인간 중심의 대화형 심리 치료 과정을 흥내 낼 수 있게 존 매카시 John McCarthy<sup>09</sup>가 만든 엘리자 Eliza 프로그램에서 아이디어를 얻은 ‘doctor 모드’<sup>10</sup>라든지, 문서의 내용을 뒤죽박죽으로 만들어 읽기 힘들게 하거나 재미있게 볼 수 있게 한 ‘dissociated-press 모드’<sup>11</sup>, 하노이 탑 문제를 애니메이션으로 보여주는

'hanoi 모드'<sup>12</sup> 등이 그것이다. 바로 이러한 깊이와 풍부함이 나로 하여금 GNU 이 맥스의 매뉴얼과 소스 코드를 읽고 더 많이 알고 싶게 만들었다.

매뉴얼의 마지막 장에는 'GNU 선언문'이라는 글이 포함되어 있었는데, 나는 이 글을 통해 이맥스를 사용하면서 느낀 '왜 이렇게 좋은 프로그램을 자유롭게 재배포하는 소프트웨어(다른 말로 오픈 소스<sup>13</sup>)로 이용할 수 있는 걸까?'라는 풀리지 않던 의문에 대한 답을 얻을 수 있었다.

리처드 스톤먼은 이 평범한 질문에 다음과 같이 대답한다.

#### 「왜 GNU를 개발해야만 했는가?」

어떤 프로그램을 좋아한다면 당연히 그것을 좋아하는 사람들과 함께 나누는 것이 황금률(대우받고자 하는 대로 대하라 – 성경<sup>14</sup>)이라고 생각한다. 소프트웨어를 판매하는 사람들은 사용자를 각각 구분하고 사용자 위에 군림하고, 사용자 서로가 프로그램을 공유하는 것을 막고자 한다. 나는 이런 식으로 사용자 사이의 결속이 깨지는 것을 거부한다..」

위의 문구 외에도 GNU 선언문에는 너무도 많은 담론이 들어 있지만 이곳에 모두 인용할 수는 없다(<http://www.gnu.org/gnu manifesto.ko.html>에서 전체 글을 참고할 수 있다). GNU 선언문은 겉으로 볼 때 사회주의자의 항변으로 해석되기 충분하다. 그러나 나는 다른 무언가를 보았는데 그것은 감춰진 사업 계획이었다. 기본 아이디어는 간단했다. 오픈 소스는 전 세계 프로그래머들의 노력을 하나로 응집할 수 있다. 또한 그렇게 만들어진 소프트웨어에 기반을 둔 (맞춤변경이나 기능 향상, 오류 수정, 사후 지원 등의) 상업 서비스를 제공하는 회사는 오픈 소스라는 이 새로운 형태의 소프트웨어에 대한 폭넓은 지지 기반과 규모의 경제를 이용할 수 있는 것이다.

자유 소프트웨어 재단의 프로그램 중에서 마음을 흔들 만큼 매력적인 것이 비단 이맥스 뿐은 아니었다. GNU 디버거, 즉 GDB는 DEC(현재는 컴팩으로 합병되었

다<sup>15</sup>)와 썬 마이크로시스템즈<sup>16</sup>가 제공하던 디버거들이 이맥스와 같이 복잡한 프로그램을 디버깅하기에 역부족이었기 때문에 리처드 스톤먼이 직접 만든 것이었다. GDB는 프로그래머가 필요로 하는 다양한 명령어와 확장 기능을 가져 큰 규모의 작업도 산뜻하게 처리할 수 있었고, 오픈 소스 소프트웨어기 때문에 프로그래머가 기능을 추가하며 더 강력한 프로그램이 되어갔다. 바로 이것이 사유 소프트웨어 proprietary software<sup>17</sup>에는 존재하지 않던 일종의 확장성이었다.

좀 더 극적인 사건은 1987년 6월에 발생했다.<sup>18</sup> 리처드 스톤먼이 GNU C 컴파일러, 즉 GCC 1.0판<sup>version</sup>을 발표한 것이다. 나는 즉시 GCC를 다운로드한 뒤에 11만 행에 달하는 코드를 빨리 습득하기 위해 이맥스와 GDB 매뉴얼에서 읽었던 모든 기법을 사용했다. 스톤먼이 발표한 첫 번째 판의 GCC는 구형 VAX와 신형 Sun-3 워크스테이션 두 개 플랫폼을 지원했는데, 두 회사<sup>19</sup>가 제공하는 컴파일러 보다 더 나은 코드를 쉽게 만들어 냈다. 나는 2주 동안 GCC를 내셔널 세미컨덕터 National Semiconductor의 신형 32032 마이크로프로세서에 이식했는데, 내셔널이 제공한 사유 소프트웨어 컴파일러보다 20%나 빠른 성능을 보여주었다. 다시 2주의 해킹을 통해 나는 성능 차이를 40%까지 벌일 수 있었다. (모토로라 68020과 경쟁하는 데 필요한 속도가 1MIPS<sup>ips</sup><sup>20</sup>인데 반해서, 막상 출시되었을 때 애플리케이션 벤치마크로 알아본 실제 속도는 0.75MIPS 밖에 되지 않았기 때문 내셔널의 칩이 사라지게 되었다고 흔히 이야기한다. GCC를 사용하면 ‘ $140\% \times 0.75\text{MIPS} = 1.05\text{MIPS}$ ’이므로 1MIPS 넘게 성능을 올릴 수 있다는 점을 생각해보자. 형편없는 컴파일러 때문에 내셔널은 얼마나 많은 대가를 치른 것인가?) 컴파일러와 디버거 그리고 편집기는 프로그래머가 매일 사용하는 세 가지 주된 개발 도구다. 그리고 GCC와 GDB, 이맥스는 사유 소프트웨어보다 월등한 성능을 갖고 있었다. 나는 사유 기술을 그보다 더 좋을 뿐 아니라 빠르게 향상되고 있는 기술로 대체하면, (경제적 이익은 물론이고) 거기에 얼마나 많은 돈이 있을지 생각하지 않을 수 없었다.

다시 한번 GNU 선언문을 인용해보자.

「유해한 수단을 사용하지 않는다면, 노동에 대한 보수와 자신의 소득이 극대화되기를 바라는 것에 아무런 문제가 없다. 그러나 지금까지 소프트웨어 산업에서 보편화된 수단은 유해한 방법이었다.

프로그램을 사용하는 것에 제한을 두어 돈을 버는 행위는 프로그램이 사용되는 범위와 방식을 제한하기 때문에 유해한 것이다. 이는 사람들이 프로그램으로부터 얻을 수 있는 인간적인 풍요로움을 전체적으로 감소시키는 것이다. 프로그램의 자유로운 사용에 대한 제한은 결국 유해한 파괴 행위라고 할 수 있다.

선량한 시민이라면 자신이 좀 더 부유해지기 위해 그러한 수단을 사용하지 않는다. 그 까닭은 만일 모든 사람이 그렇게 한다면 상호 간의 유해한 행위로 인해 결과적으로 우리 모두가 더욱 빈곤해질 것이기 때문이다.」

무거운 주제를 다루지만, 궁극적으로 GNU 선언문은 이성적인 문서다. GNU 선언문은 소프트웨어와 프로그래밍의 본질 그리고 학문 교육의 위대한 전통을 자세히 분석하여 금전적 가치에 관계없이 자신이 공유할 수 있었던 정보를 다른 사람과도 공유해야 한다는 윤리·도덕적 명령에 대해 결론을 내린다. 그러나 나는 다른 결론에 도달했다. 스톨먼과 내가 종종 논쟁하던 것 중 하나지만 소프트웨어를 자유롭게 사용하고, 개작하고, 배포하는 자유는 그러한 자유를 제한하려는 어떤 제도에 대항해 보편화되는데, 그것은 윤리적 이유가 아닌 시장 주도적 경쟁 논리에 의해 이루어진다는 점이다.

먼저 나는 스토먼이 자신의 주장을 만든 방식처럼 핵심 사항에 대해 나 자신의 주장을 만들려고 노력했다. 나는 공유의 자유가 어떻게 더 낮은 비용으로 더 큰 혁신을 이끌어 내고 더 많은 공개 표준 등을 통해 더 큰 규모의 경제를 이룰 수 있는지에 대해 설명했는데, 사람들은 예외 없이 이렇게 대답했다. “그건 참 좋은 아이디어야. 하지만 결코 그렇게 되지 않을 거야. 왜냐하면 자유(무료) 소프트웨어를 돈을 주고

살 사람은 아무도 없기 때문이야.” 나는 내 주장을 정돈하고 표현을 다듬으며 나를 초청한 사람들에게 내 생각을 전하기 위해 전 세계를 다녔지만, ”그건 참 좋은 아이디어야. 하지만…”이라는 반응<sup>21</sup>에서 결코 더 나아가지 못했다. 그렇게 2년이 지난 뒤에 나는 내 두 번째 통찰을 갖게 되었다. 모든 사람이 대단한 아이디어라고 생각 하지만 또한 실현할 수 있다고 생각하는 사람이 아무도 없다면, 내겐 경쟁자가 없는 것이다!

$$-F = -ma$$

— 아이작 뉴턴<sup>22</sup>

뉴턴의 법칙을 위와 같이 기술한 물리 교과서를 본 사람은 아무도 없을 것이다. 그러나 수학적으로 볼 때 이것은 ‘ $F = ma$ ’와 마찬가지로 타당하다. 이 접근 방식의 핵심은 방정식 자체에 집중한다면 양변을 부정한 뒤의 모습이 생소해 보일지 몰라도 방정식을 타당하게 유지할 수 있다는 것이다. 나는 오픈 소스 소프트웨어에 상업 지원을 제공하는 사업 모델이 불가능한 것처럼 보이는 이유는 사람들이 마이너스 부호에 너무 흥분한 나머지 양변의 부호를 같이 없애버릴 생각을 잊었기 때문이라고 믿었다.

군대의 침략에는 대항할 수 있지만, 때를 만난 아이디어에는 대항할 수 없다.

— 빅토르 위고<sup>23</sup>

스탠퍼드 대학의 Ph.D. 과정을 그만두고 사업을 시작하기 위해서는 내가 답을 찾아야만 하는 (매우 가설적인) 마지막 질문 하나가 남아 있었다. 먼저 내게 돈이 한 푼도 없는 것이 아니라, 반대로 창업하려고 하는 분야의 기술 중 어떠한 독점 기술도 매수할 수 있는 충분한 돈이 있다고 가정해 보았다. 나는 썬의 기술을 생각해 보았고 DEC의 기술을 생각해 보았다. 또한 내가 아는 다른 기술에 대해서도 생각해 보았다. 만약 그렇게 사업을 시작한다면 GNU를 기반으로 사업을 만든 다른 누군

가가 나를 밀어내기 전에 성공하려면 얼마의 시간이 필요할까? 초기 투자비는 건질 수 있을까?

오픈 소스 소프트웨어와 경쟁하는 것이 얼마나 불리한 것인지를 깨달았을 때, 나는 오픈 소스 사업 모델이 때를 만난 아이디어라는 것을 알게 되었다.

이론과 실제의 차이가 이론에서는 매우 작아 보이지만,  
실제에서는 참으로 매우 큰 것이다.

— 작자 미상<sup>24</sup>

이제 오픈 소스 사업 모델을 뒷받침하는 이론과 우리가 이 이론을 현실로 만들기 위해 시도한 방법들을 구체적으로 말해보려고 한다.

먼저 몇 개의 유명한 통찰로부터 시작해 보자.

자유 시장은 최대 가치를 창출하기 위해,  
자원을 가장 효율적으로 사용하며 스스로 조직화된다.

— 애덤 스미스<sup>25</sup>

정보는 만들어지는 데 사용된 비용에 상관없이,  
무료나 그에 준하는 비용으로 복제되고 공유될 수 있어야 한다.

— 토머스 제퍼슨<sup>26</sup>

자유 시장 경제의 개념은 너무 방대한 것이어서 매년 노벨 경제학상 수상 시기가 돌아오면 나는 애덤 스미스를 다른 방식으로 가장 설득력 있게 설명한 경제학자에게 상이 돌아갈 것이라고 농담 삼아 말하곤 한다. 그러나 이 농담 안에는 핵심이 있다. 소프트웨어에 좀 더 이상적인 자유 시장 체제는 아직 발현되지 않은 무한한 경제적 잠재력을 깨워 낼 수 있는 것이다.

애덤 스미스 시절의 자유 시장 경제는 개인이 여행하거나 교역할 수 있는 수준에서 가능했으며 큰 규모의 교역, 특히 국가 간의 무역은 엄격히 통제되었다. 따라서 충분히 많은 수의 상인이 당시의 지배적이던 특허 사용료 royalty 기반 제도에 환멸을 느끼게 되었을 때, 그들은 반란을 일으켜 이전의 그 어떤 정부보다 그들의 사업에 관여하지 않을 새로운 정부를 만들어 냈던 것이다. 실제로 자유는 미국 헌법의 근본적인 이상과 뼈대가 되었으며, 오늘날 국제 정치와 경제 영역에서 일어나는 모든 중요한 논제와 행동에도 자유가 근본 문제로 작용하는 것처럼 보인다. 무엇이 자유를 이렇게 중요하게 만들었을까? 자유가 경제적 번영을 책임지는 가장 큰 요인이 된 이유는 무엇일까? 이 문제에 대해 간단히 살펴보자.

문제의 오류를 더 깊이 이해할수록, 그 문제는 더욱 가치 있는 것이 된다.

— 켈빈 경<sup>27</sup>

1989년에 프로그래머가 사용할 수 있는 자유 소프트웨어 개발 도구는 분명 혁씬 없는 수준이었다. 첫째, 기초적인 기능만 제공되는 도구였다. 둘째, 이용할 수 있는 기능도 내장 한계 built-in limitation로 인해 프로젝트가 복잡해지기 시작하면 작동하지 않는 경우가 흔했다. 셋째, 해당 업체의 지원이 너무 열악했다. 따라서 대량의 하드웨어 구매나 대규모 소프트웨어 사이트 이용허락 site license<sup>28</sup> 갱신을 진행 중이어서 구매자의 지위를 이용할 수 있는 경우가 아니라면 내장 한계로 인한 문제에 당면했을 때 해결할 방법이 없었다. 마지막으로 모든 업체가 자신만의 확장을 구현했기 때문에 한 플랫폼을 일단 사용하기 시작하면 처음엔 알 수 없지만 점차 해당 플랫폼에 묶여 빠져나갈 수 없게 되어버렸다. 이 모든 것으로부터 자명하게 알 수 있듯이 자유 시장 경제의 미덕이 무엇이었던 간에 그것은 소프트웨어 시장에서 작동하지 않았다. 자유 소프트웨어 모델에 대한 검토는 그것이 봉괴된 모델이라는 것을 확인한 가치가 있었다.

오늘날에도 자유 시장 경제는 자유 소프트웨어 회사의 장벽 안에 갇혀 있다(개발

팀과 제품팀은 자금과 지원을 얻기 위해 경쟁하는 다른 팀들과 안에서 서로 분투한다). 그러나 장벽 밖에서는 소프트웨어의 사용 및 배포가 이용허락과 특히 그리고 영업비밀에 의해 엄격히 통제된다. 사람들은 자유 소프트웨어 회사가 행사한 자유로 인해 시장에서 어떤 능률과 힘이 사라진 건지 거시적이 아닌 미시적인 차원에서 궁금해할 수 있을 뿐이다. 우리는 소스 코드 수준에서 고객 지원을 제공하는 회사를 시작함으로써 그 답을 찾으려 했다.

발명은 1%의 영감과 99%의 노력으로 이루어진다.

— 토마스 에디슨<sup>29</sup>

소프트웨어 회사의 관점을 극단적으로 말하면, 일단 사람들이 구입하기 원하는 소프트웨어를 만들었다면 해당 소프트웨어의 복제물을 만들어 배포하는 행위는 화폐를 찍어내는 것과 다르지 않다는 것이다. 상품 원가는 무시할 수준이지만 매출 이익은 거의 완벽하다. 1980년대에 소프트웨어가 놓인 상황이 이런 열악한 상태를 벗어나지 못했던 이유 중 하나는 일단 화폐가 유통되기 시작하면 실제로 어떤 일이 벌어질 지에는 관심을 두지 않은 채, 화폐를 찍어내는 것 같은 추상적 모델을 완성하는 데만 집중했기 때문이라고 나는 믿는다. 소프트웨어에 대한 지원 개념은 제품 공정에서 생겨난 어떤 결합의 부산물을 없애려는 것처럼 보였고 또한 소프트웨어 지원 투자를 최소화함으로써 소프트웨어 업체는 이익을 극대화할 수 있었다.

그러나 이것은 사용자에게 불만을 줄 뿐 아니라 소프트웨어에도 마찬가지로 해롭다. 쉽게 구현할 수 있는 기능은 ‘전략상 중요하지 않다’는 이유로 대개 채택되지 않았다. 소스 코드에 접근할 수 있으면 소비자가 직접 구현할 수 있는 기능도 소스 코드를 이용할 수 없기 때문에 실속 없는 논쟁으로 남았다. 또한 궁극적으로 고객이 아닌 소프트웨어 업체와 (그들의 마케팅 부서는) 표현하기 쉽지만 쓸모는 없는 무수히 많은 기능으로 경쟁 무대를 한정시켜 버렸다.

진리는 아무도 독점할 수 없다.

— 작자 미상

보통법은 모든 사람에게 똑같이 자유로운 법전이다.

— 마이클 티만

이 세상을 더 나은 곳으로 만들 방법에 대해 멋진 이론을 갖는 것은 무척 좋은 일이다. 그러나 멋진 이론이 자생할 수 있을 때까지 필요한 자금을 조달하는 것은 전혀 별개의 문제다. 이런 문제에 직면했을 때 참고할 수 있는 서비스 기반 회사가 소프트웨어 제품 세계에는 거의 없었지만 다른 분야에는 검토할 만한 사례가 많이 있었다.

미국에서의 (또는 영국에서의) 법률 실무를 한번 생각해보자. 보통법<sup>30</sup>은 원하는 누구나 자유롭게 이용할 수 있다. 어떤 사람도 ‘로 대 웨이드 Roe v. Wade’<sup>31</sup> 사건 판결을 자신의 주장에 인용하기 위해 이용허락을 받을 필요가 없다. 실제로 일단 판결이 선고된 뒤에는 소송 비용이 얼마였든 간에 모든 사람이 그 판례를 자유롭게 이용할 수 있다. 그러나 이 모든 자유에도 불구하고 변호사는 가장 값비싼 비용을 지불해야 하는 전문가 중 하나다. 직접 사유하는 법전이 없음에도 불구하고 법률 실무는 어떻게 이런 가치를 요구하는 것일까?

사람들이 그렇게 높은 가치를 부여하는 것은 소송 과정에서의 법률 실무뿐 아니라 그를 통해 축적된 가치에 있다. 만약 여러분이 유능한 변호사를 선임할 수 있다면, 유리한 판결을 얻을 수 있고 그 판례는 법의 새로운 일부가 된다. 정의는 눈면 것이 아니라 역사로 가득 차 있는 것이다.

이것은 오픈 소스 소프트웨어로 표준을 만들고 유지관리 하는 개념과 다소 유사하다. 표준을 만들고 이를 올바르게 유지하는 데는 매우 큰 비용이 요구된다. 그러나 표준 없이 작업하거나 표준이 엉터리일 경우에 그것을 유지관리 하려 한다면 더욱 값비싼 비용이 요구된다. 따라서 유능한 사람들로 하여금 미래 표준을 확립하는 선

례를 가진 소프트웨어에 대한 작업을 하도록 하는 것은 대단히 가치 있는 일이다. 우리는 사람들이 이러한 가치 제안을 이해하고 소프트웨어 세계의 사실상 표준이 될 고품질 오픈 소스 프로그램을 만들 투자 기회를 높이 평가할 것으로 믿었다.<sup>32</sup>

## 초창기의 시그너스

이론을 준비했으니, 이제 실천에 옮길 때였다. 사업에 대해 조금이라도 안다면 서비스 기반 회사를 만드는 것은 무척 쉬운 일이다. 그러나 유감스럽게도 시그너스의 공동 창업자 세 명은 모두 사업 경험이 없는 사람들이었다.

항상 새로운 실수를 하라.

— 에스터 딘이슨<sup>33</sup>

우리는 놀로 출판사 Nolo Press<sup>34</sup>에서 나온 책을 참고해서 정관을 만들고 그 밖의 여러 요건을 갖추며 회사를 설립했다. 그러나 첫해에 아낀 돈은 나중에 푼돈까지 합쳐 수천 배의 비용 지출로 돌아왔다(우리가 받은 첫 번째 전문 자문은 시간당 수백 달러짜리였지만<sup>35</sup>, 잘못을 바로잡는데 여전히 수만 달러가 들어가고 있다. 그러나 대부분의 경우는, 우리가 자문을 구하려 했던 변호사들이 특별히 무능했다기보다 당시의 우리가 법률과 회사 문제에 대해 올바로 판단하고 그에 대한 적절한 조언을 구할 능력이 부족했다고 할 수 있기 때문에 더 나은 법률 자문을 구할 수 있었을지는 명확하지 않다).

전혀 새로운 사업 모델을 만들었기 때문에 우리는 금융과 회계, 마케팅, 판매, 고객 정보 그리고 지원 서비스 개념 또한 새로 만들었다. 사업 첫 해는 이런 일들을 처리하며 꼬박 보냈는데, 모든 것이 혼란스러웠고 모두가 사업을 정상화 시키기 위해 필요한 일을 닥치는 데로 했다. 그러나 사업이 진행됨에 따라 모든 것은 완전히 재편되어야 했다.

시그너스, 우리는 자유 소프트웨어를 구매할 수 있게 한다.

— 존 길모어<sup>36</sup>

이러한 혼란과 싸우기 위해 우리는 기본적인 사업 전제를 최대한 단순화시키려고 노력했다. 검증된 기술 소프트웨어에 대해 검증된 기술 지원을 제공하려고 했으며, 수익을 만들기 위해 규모의 경제를 이용하려고 했다. 또한 우리가 계산한 판단을 통해 고객사 내부 인력이 직접 하는 것보다 2배에서 4배의 지원 품질과 개발 능력을 제공하면서도 서비스 비용은 1/2에서 1/4로 낮추려고 했다. 오픈 소스 소프트웨어와 관련된 다른 사업들은 신경 쓰지 않았는데, 그 이유는 아직 상품성이 보이지 않았기 때문이었다. 우리는 고객에게 더 좋은 개발 도구를 더 저렴한 비용으로 공급하는 데만 초점을 맞추었으며, 계약이 거듭됨에 따라 요령을 갖게 됐다.

첫 번째 계약은 1990년 2월에 이루어졌다. 그리고 4월이 끝나갈 무렵에는 15만 달러(1990년 화폐가치 약 1억 원, 2013년 화폐가치 약 2억 4천만 원) 이상의 계약을 성사시킬 수 있었다. 5월에는 우리의 지원 서비스에 관심을 가질 것으로 판단한 50군데 예상 고객사에 홍보 편지를 보냈고, 6월에는 또 다른 100군데에 편지를 보냈다. 어느새 사업은 현실이 되어 있었다. 사업 첫해가 끝날 때까지 우리는 72만 5천 달러(1990년 화폐가치 약 5억 1천만 원, 2013년 화폐가치 약 12억 2천만 원)의 개발 및 지원 계약을 맺었고 우리가 바라보는 모든 곳에 더 많은 기회가 있었다.

그러나 이 모든 성공에도 우리는 상당히 심각한 문제를 키우고 있었다. 만약 우리가 고객사 내부 인력이 직접 수행할 경우보다 1/2에서 1/4의 비용으로 서비스를 판매했다면, 실제로는 150만 달러에서 300만 달러(1990년 화폐가치 약 10~21억 원, 2013년 화폐가치 약 24~50억 원) 규모의 계약을 맺은 것이었다. 그러나 우리는 아직 5명뿐이었다. 1명의 영업 담당과 1명의 시간제 근무 대학원생 그리고 3명의 공동 창업자가 네트워크 회선 작업부터 서류 양식 검토까지 모든 일을 다 했던 것이다. 규모의 경제가 실제로 작동하려면 사업이 얼마나 더 커져야 할까? 현재의

성장률이라면 그 시점까지 얼마나 더 많은 밤샘 근무자를 고용해야 하는 걸까? 아무도 알 수 없었다. 왜냐하면 우리에겐 어떠한 금융 모델이나 운영 모델도 없었기 때문이다.

## GNUPro

우리는 일과 인력 사이의 불균형이 실제로 심각한 문제가 되기 전에 규모의 경제를 이뤄야 한다는 결론을 내렸다. 그리고 기술자답게 생각해서 규모의 경제를 이루는 가장 빠른 방법은 꽤 많은 양을 판매할 수 있는 제품이 될 가장 작은 오픈 소스 기술 모음에만 집중하는 것이라고 결정했다. 우리는 범위를 더 좁게 잡을수록 규모의 개념을 달성하는 게 더 쉬워질 것으로 판단했다.

먼저 튼튼한 기반을 확립하라.

— 손자<sup>37</sup>

셀 도구와 파일 유트리티, 소스 코드 제어 소프트웨어에 대한 지원 계획과 심지어 인텔 386 아키텍처에 맞는 자유 커널을 만들려던 계획을 모두 버리고<sup>38</sup> 우리는 GNU 컴파일러와 디버거를 슈링크랩(shrink-wrap)<sup>39</sup> 제품으로 팔기로 결정했다. 당시에는 32비트 컴파일러를 파는 12개 정도의 서드파티(third-party) 업체<sup>40</sup>가 있었고 썬이나 HP, IBM 등과 같이 컴퓨터 시스템에 컴파일러를 함께 제공하는 또 다른 12개 정도의 업체가 있었다.

업체 수가 많아질수록, 만약 우리가 32비트 컴파일러 시장을 제패할 수 있다면 처음 계획했던 다른 멋진 일들을 모두 충분히 할 수 있을 만큼 (IBM 시스템에 대한 EDS 아웃소싱 모델<sup>41</sup>과 유사하게 오픈 소스에만 최대한 전념할 수 있게) 사업이 성장할 것으로 생각했다.

GNU 컴파일러는 이미 수십 개 호스트 환경과 12개가 넘는 타깃 아키텍처를 지원

하는 가장 폭넓게 이식된 컴파일러 중 하나였다(그 중 6개의 이식은 내가 한 것이다). GNU 디버거는 5개 정도의 네이티브 플랫폼에서 동작했으며 몇몇 사람은 임베디드 시스템을 지원하기 위한 용도로 개작하기도 했다.<sup>42</sup> 우리는 여러 프로그램을 모아 하나의 배포판에 넣고 그 밖의 구성품을 만들고 안내 문서와 설치 스크립트를 작성해 추가하고 테스트를 거쳐 포장해 출하하는 단순한 작업으로 슈링크랩 제품을 만들 수 있을 것으로 생각했다. 그러나 실제는 훨씬 더 어려웠다.

그 이유는 첫째, GCC가 1.42판에서 2.0판으로 옮겨가는 과정이었다. GCC 1.x판은 모토로라 68000이나 VAX와 같은 CISC 시스템에서 돌아가는 대부분의 다른 컴파일러를 제압하기에 충분했지만 RISC 리스크 플랫폼에서 경쟁력을 갖기 위해서는 새로운 최적화가 많이 필요했다. 내가 SPARC 스팩<sup>43</sup>으로 처음 GCC를 이식한 1988년에는 GCC가 썬의 컴파일러보다 20% 정도 느렸다. 나는 1989년에 인스트럭션 스케줄러<sup>instruction scheduler<sup>44</sup></sup>를 만들어 차이를 10%로 좁혔으며 같은 해 브랜치 스케줄러<sup>branch scheduler<sup>45</sup></sup>를 추가해 인스트럭션 스케줄러와 함께 속도 차를 5% 이내로 줄였다. CISC에서 RISC로 전환되는 세계적인 추세<sup>46</sup>와 함께 우리는 거의 모든 측면에서 최고의 컴파일러를 손쉽게 가질 수 있던 상황에서 고객의 평가를 거쳐야 하는 더 복잡한 상황에 놓이게 됐다. 간단하고 쉬운 판매는 더 이상 없었다.

둘째로 GNU C++의 개발이 미진한 상태였다. 나는 1987년 가을에 GNU C++를 만들어 첫 번째 네이티브 C++ 컴파일러를 세상에 선보였다. C++는 C보다 훨씬 복잡한 언어였고 우리가 시그너스를 시작할 때도 여전히 발전하는 언어였다.<sup>47</sup> 1990년에는 새롭고 한층 더 복잡한 몇몇 기능이 C++의 표준이 된데다 시그너스의 온갖 일들로 인해 나는 더 이상 GNU C++를 최신 상태로 유지할 시간이 없었다.

셋째로 GDB의 변종이 너무 많았다. GCC와 G++<sup>48</sup>가 중앙 배포 사이트를 통해 정기적으로 발표되는 상당히 일관된 체계였던 데 반해 GDB는 파편화되어 있었다. 오픈 소스 반대론자들은 오픈 소스 소프트웨어가 수많은 변종으로 갈라져 그 어느 하

나도 합법적인 ‘표준’이 될 수 없지만 사유 소프트웨어에는 오직 하나의 순정판 밖에 없다는 것을 장점으로 내세운다. 그러나 이것은 전 세계 수많은 사람에 의해 자신의 필요에 맞는 프로그램으로 수정되는 파편화가 이루어질 때 이를 조정할 강력한 유지관리자가 없었기 때문이다.

넷째로 사실 우리는 바이너리 작업을 할 때 연쇄적으로 사용할 완벽한 소프트웨어 모음인 툴체인<sup>49</sup>이 없었다. 어셈블러 GAS와 링커 GLD<sup>50</sup> 그리고 (binutils라는 이름으로 알려진) 다른 바이너리 유ти리티는 GCC와 GDB가 지원하는 플랫폼 중 대부분이 아닌 일부에서만 사용할 수 있었다. 그때까지만 해도 GCC가 지원하는 것 중에서 GDB도 지원하고, 다시 GAS와 GLD도 지원하는 것을 고르는 식으로 범위를 좁혀가면 동일한 소스 코드 기반에서 작업할 수 있는 플랫폼은 거의 존재하지 않았다.

다섯 번째로, 우리는 C 라이브러리가 없었다.<sup>51</sup> 이것은 썬이나 HP와 같은 네이티브 플랫폼에서는 별 문제가 안되지만, 독립 실행 애플리케이션<sup>standalone applications</sup>을 만들기 위해 해당 기능이 필요한 임베디드 시스템 개발자에게는 매우 큰 문제였다.

여섯째로 우리의 경쟁자들은 우리의 적기 생산<sup>just-in-time</sup> 기술에 필적할 만한 것이 없었지만, 각자의 틈새시장에서 매우 효과적으로 팔 수 있는 완제품이 이미 있었다. 우리는 정교한 측면 공격 계획을 우리보다 10배에서 100배는 더 큰 매출을 내는 회사들에 대한 정면 공격 계획인 슈링크랩 제품 판매로 변경했다.

마지막으로 우리들 자신의 확신이 문제였다. 빠르게 발전하는 많은 도구를 하나로 통합하는 주체가 되었을 때의 좋은 점은 우리에게 오는 서비스의 수요가 매우 분명하다는 것이다. 그러나 슈링크랩 오픈 소스 제품의 개념 자체에 대한 회의론이 제기되었다. 만약 우리가 양질의 제품을 생산하게 되면 지원 서비스는 더 이상 필요 없어질 것이기 때문에 6개월 안에 사업을 접게 되리라는 주장이었다. 이것은 그 후 4년 동안 내가 들었던 우리 사업에 대한 회의론이었다.

세상은 넘을 수 없는 기회로 가득 차 있다.

— 요기 베라<sup>52</sup>

결국 우리는 최초의 계획을 계속 추진해 나가는 것밖에 다른 방법이 없었다. 그래서 해야 할 일의 초기 6개월 치 추정을 바탕으로 우리 모두 두 사람 둷의 일을 해서 계획을 가능하게 만들기로 했다. 나는 낮에는 매출 증가를 위해 일하고 밤에는 GCC 2.0과 G++를 완성하는 작업을 도왔다. 검비Gumby라는 별명으로 알려진 두 번째 공동 창업자 데이비드 헨켈-월리스David Henkel-Wallace는 고객 지원 이사Director of Support와 최고재무책임자 CFOChief Financial Officer 직무에 더해 binutils와 라이브러리 개발 작업을 수행했다. 세 번째 공동 창업자 존 길모어John Gilmore는 GDB 작업을 맡았다. 우리는 사원을 몇 명 새로 고용했는데 그들은 첫 번째로 CVS(오픈 소스 소스 코드 관리 소프트웨어)에 모든 작업물을 넣고, 두 번째로 슈링크랩 제품이 작동 가능할 수 있는 수백 개 플랫폼을 처리할 수 있게 설정 및 설치 스크립트를 작성하고, 세 번째로 테스트 절차를 자동화하고, 네 번째로 한계에 달하도록 늘어나던 새로운 개발 계약 체결 작업을 도와주었다.

6개월 후에는 설명할 수 없을 정도로 일이 많이 진척됐는데, 몇몇 사람은 우리의 엄격한(몇몇 사람은 제한적이라고 말하는) 제품 범위에 지루해했다. 우리가 쏟은 기술과 판매 노력의 대부분은 GNU 제품<sup>53</sup>에 대한 것이었지만, (네트워크 보안 소프트웨어) 커베로스Kerberos<sup>54</sup>와 이맥스 그리고 심지어는 (그 당시 아직 개발 중이던) 우리가 사용하던 버그 추적 및 테스트 프레임워크 소프트웨어<sup>55</sup>와 같은 다른 기술 계약도 판매했다.

존은 자신이 GDB의 새로운 유지관리자가 되겠다고 인터넷에 알렸다. 그리고 만약 다음 판의 GDB에 포함되길 원하는 기능이 있다면 갖고 있는 GDB 소스 코드를 보내주면 통합할 방법을 찾아보겠다고 했다. 그 후 6주일 동안 그는 137개의 GDB를 모았는데 대부분 3.5판을 해킹한 것이었고 모두 한 개나 그 이상의 통합할 만한 기

능이 있었다. 존은 그 기능들을 모두 지원하기 위해서 GDB 4.0의 구조를 설계하기 시작했다. 나는 그 계획에 반대할 수 없었다.

한편 검비는 모든 바이너리 파일 유ти리티가 알려진 모든 오브젝트 파일과 디버깅 포맷을 처리할 수 있는 동일한 라이브러리를 사용해야 한다고 결정했다. 컴파일러와 연동하는 다양한 도구들의 상관관계를 생각해 보면 이 결정은 분명 올바른 것이었다.

개발 도구	입력 포맷	출력 포맷 <sup>57</sup>
컴파일러	아스키	아스키
어셈블러	아스키	바이너리
아카이버	바이너리	바이너리
링커	바이너리	바이너리
Size	바이너리	바이너리
Strip	바이너리	바이너리
Nm	바이너리	바이너리
디버거	바이너리	없음

각각의 도구는 바이너리 파일을 읽거나 쓰기 위한 자체적인 구현 방식이 있었으며 구현 방식들은 a.out, b.out, coff, ecoff, elf, ieee695 등의 포맷<sup>57</sup>마다 지원 수준이 각각 달랐다. 게다가 각각의 도구는 설정될 때 오직 한 종류의 바이너리 파일 포맷만 지원할 수 있게 되어 있었다. 따라서 어셈블러를 m68k-a.out 용으로 설정하면 다른 모든 도구도 a.out에 맞게 만들거나 오브젝트 파일에 독립적인 형태로 변경해야 했다. 즉, 도구가 설정된 방식에 의존적이었기 때문에 하나의 도구를 a.out에 맞춰 수정하면 그와 연관된 다른 도구도 a.out에 맞춰 변경해야 했다!

하나의 소스를 기반으로 관련 포맷을 모두 지원할 수 있는 단일 라이브러리 체계를 구축하면 모든 작업을 일관성 있게 처리하며 관리할 수 있기 때문에 규모의 경제에 좀 더 빨리 도달할 수 있다. 그리고 a.out 오브젝트 코드를 coff 라이브러리와 링크 시킨 뒤에 ieee695 실행 파일로 만드는 멋진 기능도 가능해진다! 검비는 라이브러리 구조에 대해 리처드 스톤먼과 토론하며 실제 디자인을 만들기 시작했다. 스톤먼은 그런 라이브러리를 구현하기 위해서는 도구 대부분을 모두 다시 만들어야 하고

유지 관리하는 것 또한 너무 힘들 것이기 때문에 어려운 작업이라고 말했다. 검비는 스톤먼에게 ‘X같이 대단한 일 Big Fucking Deal’<sup>58</sup>까지는 아니라고 대답했는데, 새로운 BFD 라이브러리의 이름은 이런 유래를 갖고 있었다(그러나 고객에게는 BFD가 Binary File Descriptor를 의미한다고 설명했다).

개발이 진행되는 동안 나는 현금 수입을 유지하기 위해 계약 판매를 계속해야만 했다. 나는 매 분기 더 높은 매출 목표를 세웠는데 더 많은 계약을 위해 더 많은 자원이 필요했지만 최고의 기술자 모두는 내가 신경 쓸 수 없는 신제품 개발에 매여 있었다. 모든 GNU 툴체인 개발이 50%를 넘기 전까지는 GNU 소프트웨어 개발을 하면 할수록 인터넷으로부터 돌아오는 것이 더 적어졌다. 이 때문에 오픈 소스 모델이 반대로 작용하는 것처럼 보이는 동안 영업과 개발 사이에 긴장이 높아졌다.

이러한 상황은 일시적인 것이 아니어서 최초의 ‘진보적 릴리스’가 마침내 완성될 때까지 자그마치 1년 6개월이라는 시간이 소요되었다! 역사적인 그날에야 비로소 나는 단일 소스 코드 기반에서 완전한 C/C++ 개발 도구를 만들 수 있다는 사실을 처음으로 확신했다. 그러나 한 팀의 해커가 Sun-3와 Sun-4 두 개의 플랫폼을 지원하는 툴체인 2개를 만든 기간보다 짧은 시간에 나 혼자 6개 GCC 이식과 3개의 GDB 이식 그리고 C++ 네이티브 컴파일러와 디버거를 단일 소스 기반에서 만들었던 과거를 생각하면 나는 할 말이 없었다!

여기에는 참작할 두 가지 사실이 있었다. 첫째, 새롭고 유용한 많은 기능과 함께 툴체인이 이전 어느 때보다 훨씬 잘 동작했다. 둘째, 단지 도구를 다시 만든 것이 아니라 설정 스크립트와 자동화된 테스트 프레임워크의 구현까지 모든 기반 작업을 직접 했기 때문에, 가상적으로 무제한의 범위에 있는 임베디드 시스템 플랫폼을 포함해서 더욱 많은 호스트/타깃 조합 지원을 기대할 수 있었다.

우리는 이 프레임워크를 테스트했는데, 대단히 성공적이었다.

일시	릴리스 이름 <sup>59</sup>	네이티브	임베디드	총 플랫폼
1992년 3월	p1	2	0	2
1992년 6월	p2	5	0	5
1992년 9월	p3	5	10	15
1992년 12월	p4	5	20	25
1993년 3월	q1	5	30	35
1993년 6월	q2	5	45	50
1993년 9월	q3	7	53	60
1993년 12월	q4	8	67	75
1994년 3월	r1	10	75	85
1994년 6월	r2	10	80	90
1994년 9월	r3	10	85	95
1994년 12월	r4	10	90	100

기술팀이 GNUPro 제품<sup>60</sup>을 만들기 위해 큰일을 하는 동안 영업마케팅팀은 이 제품의 판매 방법을 고민했다. 1991년에 우리는 어플라이드 머티어리얼즈 Applied Materials<sup>61</sup>에서 해고된 직후에 소프트웨어 영업을 배우려던 젊은 경영학과 학생 한 명을 고용했다. 그녀는 영어가 모어가 아니었음에도 불구하고 업무를 매우 빨리 파악해 나갔다. (간혹 주말마다 사무실에서 C 언어 프로그래밍을 혼자 공부하기 했지만) 그녀가 해커인 것은 아니었다. 그러나 그녀는 오픈 소스 방식에 대한 열렬한 지지자가 되었다. 6개월간의 매우 성공적인 영업 후에 그녀는 내게 영업 프레젠테이션에 참석해 달라고 말했는데, 그녀의 프레젠테이션을 보고 나는 몹시 당황했다. 나는 언제나 해커의 방식으로 오픈 소스를 팔아왔다. 주로 기술적인 장점에만 초점을 맞춘 것이다. 그러나 그녀는 우리가 하는 있는 작업의 본질적인 복잡함과 우리가 제공하는 오픈 소스 소프트웨어의 사업적 가치에 대해 설명했는데, 이것은 결과적으로 왜 고객들이 오픈 소스 작업을 직접 하는 것보다 우리 제품을 구입하는 것이 더 나은지를 설명하는 데 도움이 됐다. 나는 우리의 기술과 인력이 고객의 것보다 어떻든 더 우수하다는 사실을 내세웠지만(이것은 고객 대부분이 듣기 좋아할 만한 이야기가 아니다), 그녀는 고객의 기술자가 우리가 해놓은 기본적인 이식과 지원, 유지보수 작업으로부터 얻을 수 있는 이점을 설명했다.

결국에는 이 두 가지가 함께 똑같은 구매 요인으로 작용하여 우리는 높은 매출 성과를 이룰 수 있었다.

연도	매출액 <sup>63</sup>	수익성(%)	누적 CAGR <sup>63</sup>
1990년	725,000달러	거의 없음	없음
1991년	1,500,000달러	1	106%
1992년	2,800,000달러	2	96%
1993년	4,800,000달러	3	87%
1994년	5,700,000달러	4	67%

왓슨 군! 어서 이리와 주게!

— 알렉산더 그레이엄 벨<sup>64</sup>

이러한 노력의 결과는 새로운 중요 기술이 되어 인터넷으로 돌아가 당당한 표준이 되었다. 예를 들면 (빌드, 호스트, 타깃 플랫폼의 세 가지 독립 변수를 기반으로 소프트웨어 설치 환경을 설정할 수 있는 통합 설정 스크립트인) GNU configure와 (configure 스크립트를 자동 생성하기 위한 고수준 스크립트) autoconf, (autoconf에 의해 만들어진 환경 변수에 맞춰 makefile을 만들어 주는) automake, (회귀 검증 도구) DejaGNU, (버그 추적 및 테스트 프레임워크 소프트웨어) GNATS 등이 그것이다.<sup>65</sup>

GNUPro 툴킷은 현재 175개가 넘는 호스트/타깃 조합을 지원하는데, 이 숫자는 시장에 나와 있는 실제 상품 수에 따른 제한이지 GNUPro나 우리가 보유한 기술로 인한 것은 아니다.

사실 GNUPro의 압도적인 시장 점유율 때문에 몇몇 업체가 우리와 경쟁하기 위해 GNU 소프트웨어에 대한 상업 지원 판매를 발표한 적이 있다! 그러나 다행스럽게도 오픈 소스 모델이 다시 구원이 되었다. 우리 직원의 대부분은 우리가 지원하는 소프트웨어의 주된 저자이거나 유지관리자이기 때문에 경쟁사의 능력이 우리가 보유한 100명이 넘는 직원과 대등할 수 없다면 ‘정통 GNU 공급업체’로서의 우리의 지위를 잊어갈 수 없다(GCC와 GDB 그리고 관련 유ти리티에 대한 모든 개작의

80% 이상이 우리가 작업한 것이다). 그들이 바랄 수 있는 최대치는 고객이 원하는 추가 기능을 덧붙이는 것 정도다. 그러나 이 경우에도 그들이 작업한 소프트웨어는 오픈 소스이기 때문에 더해진 가치가 무엇이든 간에 시그너스로 돌아온다. 우리는 그 기능을 살펴 우리 제품에 통합하거나 무시할 수 있다. 승자와 패자의 두 자리를 놓고 싸우는 사유 소프트웨어와 달리 오픈 소스는 마치 뢰비우스의 띠 위에서 싸우는 것과 같아서 모든 것이 주된 유지관리자 쪽으로 흘러 들어온다. 따라서 우리의 경쟁자가 같은 GNU 공간 안에서 몇몇 전술상 이점을 갖게 된다고 해도 그것은 결국 시그너스의 이익이 된다. 1989년에 창업한 우리가 갖는 선점자로서의 이점은 경쟁에 10년 앞서 있다는 것이다.

## 도전들

132쪽 표에서 볼 수 있듯이 우리의 성장률이 인상적으로 지속되고는 있지만 점차 둔화되고 있다. 우리가 오픈 소스 소프트웨어의 가치와 장점을 팔려고 노력할 때, 회의론자와 잠재 고객들이 우리의 사업 모델에 다음과 같은 문제를 제기했다.

### 분별성<sup>66</sup>

고객이 왜 경쟁자의 이점을 위해 돈을 지불할 것인가?

### 확장성

서비스 기반 사업을 어떻게 확장할 수 있을까?

### 지속 가능성

고객이 시그너스를 필요로 할 미래에도 계속 존재할 수 있을까?

### 수익성

오픈 소스 소프트웨어로 수익을 만들 수 있을까?

### 관리 가능성

오픈 소스 소프트웨어가 균일한 품질을 갖게 지속해서 관리할 수 있을까?

### 투자 가능성

소프트웨어 지식재산권을 갖지 않은 기업에 투자자가 관심을 보일까?

다섯 명의 임베디드 시스템 프로그래머를 둔 관리자에게 1만 달러(약 1천만 원) 가격의 서비스 지원 계약을 판매하는 사업 모델로 상장 기업이 될 수 있을까?<sup>67</sup> 오픈 소스는 최상의 혁신적 소프트웨어 개발 그룹으로 가는 문을 열어주는 좋은 수단임에도 불구하고 주류 시장에서는 오히려 주된 방해물이 된다고 증명된 바 있다. 우리에게는 이때가 제프리 무어Geoffrey Moore가 그의 책에서 밝힌 ‘캐즘을 넘는다’<sup>68</sup>는 말의 의미를 체득하는 시기였다.

이러한 사실은 내가 포천 100대 기업<sup>69</sup> 중 한 곳에서 무선 통신 시스템을 만들던 개발팀을 방문했을 때 절실히 느낄 수 있었다. 그들은 품질 관리 공정의 일부로 상당히 많은 항목에 대해 그들의 제품뿐 아니라 공급업체의 제품에 대한 평가도 함께 수행했다. 공급업체 모두가 평가 항목 전부나 대부분에 ‘우수’나 ‘최우수’ 등급을 받았던 데 반해 유독 임베디드 도구 공급업체만은 품질 관리 공정이 진행된 3년 동안 매번 모든 항목이 ‘불량’이나 ‘불합격’ 등급으로 최하위였다. 그러나 우리 제품에 대한 고객들의 호평과 우월한 기술적 기능, 더 낮은 가격에도 불구하고 경영진은 잘 알려지지 않은 솔루션으로 사업을 진행하길 꺼려 우리 제품을 구입하지 않으려 했다. 나는 제품을 구입하는 데 실제로 이용하지 않을 것이라면 왜 굳이 제품을 평가하고 자료를 모으는 것인지 무척 의아해하며 그곳을 나왔는데, 그것은 잘못된 생각이었다. 그런 생각 대신 나는 그것이 주류 시장의 전형적인 행동 방식이라는 것을 깨닫고, 문제를 바로잡는 방법은 고객을 탓하는 것이 아닌 우리의 마케팅 방법과 메시지를 개선하는 것이라는 점을 알아야 했다.

그러나 우리가 가진 문제가 외부에만 있는 것은 아니었다. 많은 고객은 우리가 무슨 말을 하든 간에 우리의 현재 상태 이상으로 지원 사업을 확장하기에 충분한 인

원을 고용할 수 있을 것으로 믿지 않았다. 이런 생각은 매우 틀린 것이기도 하고 또 한 맞는 것이기도 했는데, 기술직원을 고용하는 면에서는 정말 잘못된 생각이었다. 시그너스는 기술자들이 만든 회사다. 그리고 우리의 문화와 오픈 소스 사업 모델, 그리고 세계 최고의 오픈 소스 개발팀에 합류할 기회는 우리가 고용하려던 개발자에게 항상 시그너스를 매력적으로 만들었다. 이직률을 전국 평균과 비교하면 (특히 실리콘밸리의 평균과 비교하면) 다른 기업의 대략 1/4에서 1/10 수준이다.

그러나 관리직원을 고용할 때는 전혀 다른 상황이 되었다. 우리가 접촉했던 사람 대부분은 주류 시장의 고객과 같은 많은 편견과 거부감을 가진 채 시그너스에서 일하는 것에 관심을 보이지 않았다. 그들은 시그너스에 매력을 느끼지 않았다. 시그너스에 매력을 느낀 사람은 흔히 잘못된 이유를 갖고 있었다. 우리는 기술팀의 기술직원이 50명을 넘을 때까지 두 명의 관리직원만을 구할 수 있었다.<sup>70</sup> 게다가 관리직원들이 오픈 소스가 무엇인지 이해하고 오픈 소스 회사답게 관리하려고 노력할 수록 대체로 실패해서 의사소통과 제품 공정, 경영 관리, 직원 만족도 모두가 낮아졌다.

무척 아이러니하게도 우리 또한 사업에 폐쇄 소스 요소를 수용하지 않으려는 관리직원은 뽑지 않았다. 오픈 소스는 사업 전략이지 사업 철학이 아니다. 우리는 회사의 목표 전체를 충족시키기 위해 오픈 소스 제품과 폐쇄 소스 제품을 함께 관리할 충분한 유연성이 없는 사람은 고용하고 싶지 않았다.

결국 우리는 오픈 소스 소프트웨어의 모든 의미를 바로 이해할 관리직원은 고용할 수 없다는 사실을 인정하게 되었다. 경영진은 관리직원이 실수할 것을 예상해야 만 한다(이는 실수로 인한 비용을 예산에 미리 반영해 놓아야만 한다는 것을 의미한다). 그들은 그런 실수로부터 배울 수 있어야만 한다. 경력이 있는 사람 대부분은 자신의 경험에 맞춰 일을 변경하려고 하는데, 이는 시그너스에서 일을 망치는 지름길과 같았다. 경험을 통해 빨리 배우면서 또한 경험을 통해 관리 업무를 잘 수행할 수 있는 사람을 찾는 것은 매우 어려운 일이었는데, 우리는 그런 사람이 수십 명이

나 필요했다.

오픈 소스 모델은 이 모든 문제에 대해 탁월한 회복력을 보여주었다. 형편없이 설정한 예측이나 서툰 계약 이행 때문에 때때로 고객을 잃었음에도, 화폐가치 후입선 출법<sup>71</sup>으로 산정한 연간 계약 갱신율은 1993년 이후 대략 90%를 유지했고 우리가 고객을 잃은 첫 번째 이유는 고객의 프로젝트 종결에 따른 ‘사업 철수’ 때문이었다. 다른 회사들이 쓰러져 갈 때도 두 가지 요인이 우리가 살아남도록 도와주었다. 첫 번째는 직위와 근무 연수에 상관없이 모든 임직원이 고객과의 약속을 지켜야 하는 중요성을 인식했다(그 누구도 고객 지원보다 우위에 있을 수 없었다). 두 번째는 모든 방법이 실패한 경우에도, (고객에게 소스 코드가 있기 때문에) 자신의 문제를 스스로 해결할 힘을 얻을 수 있었다. 이러한 요인을 통해 당시의 시그너스는 내부적으로 놀랄 만큼 많은 혼란이 있었음에도 소프트웨어를 납품하지 못했다는 이유로 떠나는 고객이 거의 없었는데, 이는 우리가 들은 사유 소프트웨어 경쟁업체나 지원되지 않는 오픈 소스 소프트웨어를 사용하는 사람들의 이야기와 큰 대비가 된다.

## 오픈 소스를 넘는 투자 - eCos

임베디드 업계의 현실은 비교적 소수의 회사가 반도체를 만들고 또한 비교적 소수의 OEM<sup>Outside Equipment Manufacturer</sup><sup>72</sup> 업체가 임베디드 제품을 만들기 위해 거의 대부분의 반도체를 구매한다는 것이다. 시장의 나머지는 흥미로운 제품을 만드는 매우 많은 수의 소규모 업체로 구성되는데, 이들은 새로운 칩 디자인이나 소프트웨어 솔루션을 만들어야 할 정도의 규모를 추구하지 않는다.

반도체 제조업체와 OEM 업체 사이에는 수백 개의 작은 소프트웨어 회사가 있으며 그들은 모두 독자적인 소프트웨어를 판매한다. 예를 들면 현재 시장에는 상업적으로 지원되는 120개가 넘는 RTOS<sup>Real Time Operating System</sup>가 있다. 그러나 IDC<sup>73</sup> 자료에 따르면 이 중 어느 한 업체도 6% 이상의 시장 점유율을 갖지 않는다. 이는 10년 전 유닉스 시장과 크게 다르지 않은데, 단지 파편화가 20배 더 많을 뿐이다! 이

런 파편화는 불필요한 중복과 비효율성, 가격 폭리 등과 같은 자유 시장 경제의 모든 고전적 퇴행 사례로 이어진다. 반도체 제조업체와 OEM 업체가 원하는 것은 시장 진입 시간의 단축, 즉 TTM<sup>Time To Money</sup><sup>74</sup>을 가속화하는 표준이었고 상용 RTOS 업체는 시간과 비용 중 하나 또는 둘 모두를 너무 많이 소모하고 있었다.

임베디드 시스템 시장에서 우리는 떠오르는 샷별이었다. 우리는 시장의 선두 업체 보다 두 배 빠르게 성장하고 있었고 상위 네 개 경쟁업체를 한 자릿수 성장으로 묶어두었다. 그러나 우리는 시장의 진정한 선두로 취급되지 않았고 우리 또한 그렇게 행동하지 않았다. 1995년에 우리는 주요 고객들과 임베디드 시스템 업계에서 가능한 일과 그렇지 않은 일에 대해 많은 대화를 나눈 뒤에 GNUPro 컴파일러와 디버거는 문제를 찾아 해결하는 것까지만 할 수 있다는 사실을 이해하기 시작했다. 임베디드 시장이 더 필요로 하는 것은 반도체 추상화 계층<sup>Silicon Abstraction Layer</sup><sup>75</sup>, 즉 표준 C 라이브러리나 실시간 POSIX API 밑단에 놓이는 소프트웨어 계층이었다. 우리의 제품을 확장해 더 핵심적인 방법으로 제공할 새로운 기회가 거기에 있었다.<sup>76</sup>

우리는 시장 현황을 파악해 갔다. 120개가 넘는 상용 RTOS와 1,000개가 넘는 사내<sup>in-house</sup> RTOS가 있다는 사실은 기술적인 측면에서 지금까지 그 누구도 ‘한 제품으로 모든 요건을 충족할 수 있는’ 충분히 설정 가능한 RTOS를 만들지 못했다는 것을 의미한다. 또한 사업적인 관점에서 볼 때 우리는 린타임 로열티<sup>77</sup>가 수익을 깎아 먹는다는 사실에 주목했다. 따라서 새로운 RTOS는 로열티가 없어야 한다. 바꿔 말하면 GNUPro 제품을 둘러싼 시장을 모두 아우르기 위해서는 완전히 새로운 세계적 수준의 기술을 만들어야 하고, 또한 그 제품을 무료로 제공해야 했다. 경영관리팀은 이 아이디어를 실제로 추진하는 것을 거의 1년 동안 반대했다.

그러나 일단 이 전략을 추진하기로 한 뒤부터 경영관리팀은 ‘이 제품으로 어떻게 수익을 낼 수 있을까?’라는 문제를 놓고 싸움했다. 우리가 GNUPro 주변 시장을 통합해 간다고 해도 기존 사업 모델을 어떻게 임베디드 운영체제에 다시 적용할 수

있을 지가 경영관리팀에겐 명확하지 않았다.

우리는 완전히 모순되는 상황에 당면했을 때 어떤 회사든 시도해 보는 현명한 접근 방법을 써보았다. 먼저 우리가 수익을 낼 방법을 찾았다고 가정하면, 고객의 문제를 해결하고 시장 1위가 되기 위해 해야 할 나머지 일들은 무엇일까? 첫 번째, 우리는 혁신적인 새로운 설정 기술을 개발해야 한다. 두 번째, 그 설정 기술을 통해 사용자가 무언가를 설정할 수 있는 나머지 본체를 만들어야 한다. 세 번째, 이 모든 것을 시장의 기회가 사라지기 전에 해내야 한다. 소프트웨어 개발에는 돈이 필요하다. 그리고 예정에 맞춰 제품을 출시하기 위한 소프트웨어 개발에는 더 많은 돈이 필요하다.

우리가 시그너스를 시작했을 때, 우리는 벤처 투자회사들이 결코 우리의 사업을 이해할 수 없으며, 설령 그렇게 된다고 해도 그건 그들이 우리를 위해 유익하게 해줄 것이 아무것도 없는 5년이나 그 이상이 지난 미래일 것으로 생각했다. 그러나 다행히도 이러한 가정은 모두 틀린 것이었다.

우리의 첫 번째 사외 이사 필리프 쿠루토Philippe Courtot는 1992년 초에 내게 선두 벤처 투자회사들을 소개해 주었다. 나는 그들 모두에게 우리의 사업 모델과 기술, 그리고 미래 목표에 대해 매우 솔직히 이야기했고 또한 시그너스는 자체적으로 자금을 조달하는 회사로 만든 것이기 때문에 투자를 원하지 않는다는 말도 했다. 실제로 시그너스가 매년 80%씩 성장하는 동안에 연간 1% 포인트씩 수익성을 늘려 갔다는 것은 (내가 아는 한도에서) 우리 사업이 궤도에 올랐다는 꽤 좋은 지표였다.<sup>78</sup> 벤처 투자업계의 유력한 소프트웨어 산업 분석가 로저 맥나미Roger McNamee의 말이 이를 잘 입증한다. “나는 시그너스의 사업 모델로부터 놀라움과 경탄을 금할 수 없다. 나는 오픈 소스 사업 모델이 잘 작동한다는 것에 놀랐다. 하지만 생각하면 할수록 그 사업 모델을 내가 먼저 생각해 내지 못했다는 사실이 더 놀랍다!”

스스로 당면 문제를 해결했기 때문에 외부 자금은 필요 없다고 생각하는 것은 즐거

운 일이다. 하지만 1996년의 현실에는 자기 자금 조달 방식의 GNUPro 사업을 넘어서는 훨씬 큰 기회가 있었고 우리는 새로운 계획과 동반자가 필요했다.

우리는 그레이록 매니지먼트 Greylock Management와 오거스트 캐피털 August Capital 두 군데 벤처 투자회사를 찾았다. 그들은 우리의 사업 방식과 성과를 이해했으며 올바른 경영과 지침이 있다면 우리가 무엇을 할 수 있을지 이해했다. 또한 우리 계획을 실행하기에 충분한 자본이 있었다. 그들은 625만 달러(1997년 화폐가치 약 59억 원, 2013년 화폐가치 약 96억 원)를 투자했는데, 이 금액은 1997년 상반기에 소프트웨어 회사에 대해 이루어진 가장 큰 규모의 사모 발행이었다.<sup>79</sup> 계획이 이제 본격적으로 시작되었다.

샘, 나는 그것들이 싫어. 난 초록색 계란과 햄이 싫다고.

— 닉터 수스<sup>80</sup>

기술팀이 개발에 전념하는 동안 경영관리팀은 사업 모델을 만들기 위해 계속 고민했다. 왜냐하면 처음에는 eCos Embedded Cygnus Operating System<sup>81</sup>를 상품화할 알맞은 수의 구조를 찾을 수 없었기 때문이다. 기술적인 측면에서는 이 제품 하나를 모든 개발 환경에서 사용할 수 있는 시스템 설정 능력이 핵심 요건이라는 것을 알고 있었다. 사업적인 측면에서도 마찬가지로 한 제품으로 모든 환경을 충족시키는 것이 임베디드 시스템 개발을 위해 유익하고 통일된 표준을 만들 핵심이라는 것을 알고 있었다. 그러나 우리가 여전히 풀지 못한 것은 이러한 이익을 위해 ‘누가 값을 지불하려고 할 것인가’였다. 기술팀과 경영관리팀은 각자의 문제에 대해 독립적으로 1년 반 동안 작업했다. R&D 비용은 늘어만 갔다. 오픈 소스의 역설은 조화되지 못했고 해결 방법은 보이지 않았다.

마침내 처음 구상했던 것을 기술팀이 시연할 수 있게 되었을 때, 우리가 실제로 만 들어 낸 것이 무엇인지가 경영관리팀에게 분명해졌다. 그것은 세계 최초의 오픈 소

스 아키텍처<sup>82</sup>였다. 나는 GCC를 처음 본 때만큼 흥분했다.

오픈 소스는 해커에게 모두 좋은 것이다. 그리고 오픈 소스가 표준을 만드는 방식은 최종 사용자를 위해 좋은 것이다. 그러나 오픈 소스 소프트웨어로 해커가 할 수 있는 것과 일반 사용자가 할 수 있는 것에는 차이가 있다. 우리는 eCos가 해커 공동체뿐 아니라 주류 시장의 임베디드 시스템 개발자도 끌어안을 수 있는 제품이 되기를 원했다. 우리의 착안점은 eCos의 설정과 맞춤변경 그리고 기본적인 유효성 검증을 자동화된 방식으로 수행할 수 있는 고수준 도구를 제공함으로써 당시 사내 RTOS 개발자들이 해야 했던 수작업 단계를 대체할 수 있게 하려는 것이었다. eCos를 소스 코드 수준에서 제어하는 고수준 설정 도구를 만들었고 그리고 이러한 설정 도구를 통해 관리될 수 있도록 소스 코드를 설계함으로써 우리는 단 한 줄의 C나 C++ 코드를 보거나 만들지 않아도 최종 사용자가 소스 코드 수준에서 가상적으로 작업할 수 있게 만들었다. 우리의 성공은 (거의 최소한의 반도체 추상화 계층인) 700B에서 (인터넷 통신과 파일 시스템을 갖춘 완전한 RTOS로서) 50KB를 넘는 수준까지 eCos를 확장할 수 있다는 것이 증명한다.

오픈 소스가 하나의 특성일 뿐 아니라 eCos의 기술적 원동력도 된다는 사실을 일단 깨닫게 되자, 그리고 우리 스스로 그 사실을 증명하게 되자 우리는 제품 성능 경쟁에서 10배의 이점을 갖게 되었다(오브젝트 수준의 설정성으로 10배의 공간 절약 그리고 소스 코드를 이용하는 데 있어 10배에서 100배의 프로그래밍 효율). 우리는 이러한 성능 이점을 시장에 전달하기 위해 제품 출시를 준비했는데, 시장의 사전 반응은 대단히 긍정적이었다.

GNU에 기반을 둔 우리 사업을 불가능한 것으로 여겼던 과거를 생각한다면, 이제 eCos가 시그너스 솔루션즈와 세계를 위해 만들어갈 가능성에 대해 충분히 상상할 수 있을 것이다.<sup>83</sup>

## 미래에 대한 생각과 전망

오픈 소스 소프트웨어가 기술 자유 시장의 고유한 효율성에 문을 두드리지만, 이는 상당히 유기적이고 예측 불가능한 방법 안에서 이루어지는 것이다. 오픈 소스 사업은 애덤 스미스의 ‘보이지 않는 손’의 역할을 맡아 전체 시장을 지원하며 자신의 미시 경제적 목적을 달성하고자 한다. 따라서 미래의 가장 성공적인 오픈 소스 기업은 인터넷 공동체로부터 최상의 협력을 낳는 기술을 잘 끌어내어 사용자 공동체의 가장 중요한 기술적, 사업적 도전을 풀 수 있는 업체가 될 것이다.

오픈 소스 소프트웨어로 만들어진 인터넷은 새로운 오픈 소스 소프트웨어 개발을 위한 가장 환상적인 원동력이다. 르네상스가 학문 지식을 발전시키고 이용하는 방식을 바꾼 것처럼, 사람들이 인터넷과 오픈 소스에 계속 연결될수록 우리는 소프트웨어의 발전과 이용 방식이 바뀌는 변화의 증인이 될 것이다. 오픈 소스 소프트웨어가 주는 자유가 있다면, 나는 그에 맞는 결과가 있을 것이라 기대한다!

그는 새로운 기술에 마음을 쏟았고, 그것으로 자연의 법칙을 바꾸어 나갔다.

— 제임스 조이스<sup>84</sup>

---

01 · 역사주\_이 책의 영문판 초판은 1999년 1월 10일에 출판되었다. 시그너스 솔루션즈는 이 책이 출판된 이후 인 1999년 11월 15일에 리눅스 배포업체 레드햇(Red Hat)으로 6억 7천4백만 달러(1999년 화폐가치 약 8천억 원, 통계청이 발표하는 연도별 소비자 물가 상승률을 고려한 2013년 화폐가치 약 1조 2천억 원)의 금액으로 인수·합병되었으며, 이 글의 저자 마이클 티만은 레드햇의 최고기술책임자 CTO(Chief Technical Officer)가 되었다. 2013년 현재는 레드햇의 오픈 소스 부문 부사장으로 근무하고 있다. 시그너스란 이름은 ‘Cygnus: Your GNU Support’라는 문구를 이용해 만든 재귀적 두문자어다. 마이클 티만의 말에 따르면 회사 이름으로 사용하기 위해 GNU라는 단어가 들어가도록 만든 여러 개의 이름 중에서 가장 어색하지 않은 것을 고른 것이라고 한다(이 책의 다른 글 ‘GNU 운영체제와 자유 소프트웨어 운동’ 중 <역자주 13> 참고). 시그너스 솔루션즈는 GNU 프로젝트와 직접적인 연관을 가지며 시작한 최초의 자유 소프트웨어 기업

이면서 가장 성공적인 사례에 속한다. 또한 레드햇으로의 인수·합병으로 인해 역사 속으로 사라진 기록이기 때문에 기업 사례 분석 및 학습에 참고할 수 있도록 가능한 많은 주석을 넣었다. 또한 이 번역문은 컴퓨터 관련 분야를 전공하는 대학 2~3학년을 대상 독자로 삼아 기술 사항에 대해서도 그에 맞는 주석을 넣었다. 필요할 경우 주석을 참고하며 읽어 본 뒤에 주석 없이 본문만 다시 한 번 읽어 볼 것을 권하고 싶다. 이 번역문은 등록상표나 이름을 만든 주체가 사용하는 다른 형태가 있을 경우, 외래어표기법이 아닌 해당 표기를 사용한다. 왜냐하면 자신이 만든 이름이나 등록상표는 사람들이 불러주길 원하는 자기 스스로가 확립한 정체성의 표현이기 때문이다. 예를 들어 'Sun Microsystems'의 외래어표기법에 따른 올바른 표기는 '선 마이크로시스템스'지만, 이 회사는 한국에 '썬 마이크로시스템즈'로 상표를 등록했기 때문에 이 번역문은 '썬 마이크로시스템즈'로 표기한다. 시그너스 솔루션즈의 경우에는 한국에 상표 등록을 하지는 않았지만 같은 시기에 등록한 거의 모든 회사가 (그리고 현재에도) '설루션스'가 아닌 '솔루션즈'를 사용하기 때문에 역사적인 이유에서 '솔루션즈'로 표기한다. 시그너스 솔루션즈의 과거 홈페이지는 <http://web.archive.org/web/19970605161644/http://www.cygnus.com/>에서 참고할 수 있다.

- 02 역자주\_<http://www.forbes.com/global/1998/0810/0109044s1.html> 참고.
- 03 역자주\_주식을 외부에 공개하지 않은 회사를 비공개회사(privately held company)라 한다. 비공개 회사가 기업 공개(IPO: Initial Public Offering) 절차를 통해 거래소 상장 등이 이루어지면 공개 회사(publicly-held company)가 되는데, 공개 회사는 형태와 규모에 맞게 증권거래법 등에 의한 여러 가지 규제를 받는다.
- 04 역자주\_1996년과 1997년의 순위는 각각 92위와 53위다. <http://www.bizjournals.com/sanjose/research/bol-marketing/> 참고.
- 05 역자주\_1998년 순위는 376위다. <http://web.archive.org/web/20000520095552/http://www.softwaremag.com/98sw500/sm98rnk8.htm#Cygnus> 참고.
- 06 역자주\_Department of Corporations는 캘리포니아 지역의 기업 등록과 허가 및 감독 업무를 맡은 주정부 안의 부서다. 이 명칭에는 공식적으로 사용되는 번역어가 없고, 주한 미국 대사관과 캘리포니아 총영사관을 통해서도 적절한 용어를 찾을 수 없어 2000년 종이책에서는 기업부로 번역했으나 2013년 전자책에는 법인국으로 옮긴다.
- 07 역자주\_시그너스 솔루션즈는マイ클 티만(Michael Tiemann), 존 길모어(John Gilmore), 데이비드 헨켈-월리스(David Henkel-Wallace) 세 명이 각각 2천 달러씩 6천 달러를 투자해 만든 회사다. 실리콘밸리의 많은 벤처 기업이 그런 것처럼 공동 창업자 중 한 명인 데이비드 헨켈-월리스의 방 두 칸짜리 아파트에서 사업을 시작했다. 캘리포니아주 회사법에는 법인 회사 설립을 위한 최저 자본금 규정이 없다. 따라서 여기서 말하는 6천 달러는 창업 초기 자금을 주거래 은행 계좌에 예치했다는 의미다(한국의 경우 5천만 원의 최저

자본금 제한 규정이 있었으나 창업 규제 완화를 위해 2009년 개정 상법부터 폐지되었다). 시그너스 솔루션즈의 처음 이름은 시그너스 서포트였으며 마케팅 전문 인력이 영입된 몇 년 후에 시그너스 솔루션즈로 회사 이름을 변경했다.

- 08 역자주\_자유 소프트웨어 관련 수익 사업은 1985년부터 자유 소프트웨어 재단이 맡았기 때문에 티만의 글을 더 정확히 표현하면 ‘리처드 스톤먼이 쓴 매뉴얼을 자유 소프트웨어 재단이 출판한 것이다’로 정리할 수 있다. GNU 이맥스는 이 책의 다른 글 ‘GNU 운영체제와 자유 소프트웨어 운동’ 중 ‘GNU 이맥스’ 절에 자세히 소개되어 있다. GNU 이맥스 매뉴얼은 <http://www.gnu.org/software/emacs/manual/emacs.html>에서 다운로드할 수 있다.
- 09 역자주\_원문에는 존 매카시(John McCarthy, 1927~2011)가 엘리자를 만든 것으로 되어 있지만, 그는 인공 지능 언어 리스프의 창시자지 엘리자의 저자는 아니다. 엘리자는 요제프 바이첸바움(Joseph Weizenbaum, 1923~2008)이 1966년에 만든 프로그램이다.
- 10 역자주\_요제프 바이첸바움은 엘리자를 일종의 게임으로 만들었는데, 이 프로그램이 출력하는 말을 심각하게 받아들이는 사람이 생겨나자 이를 정말로 받아들이지 말라고 해명하고 다녔다는 일화가 있다. doctor 모드로 들어가기 위해서는 이맥스를 실행한 뒤에 PC에서 일반적으로 사용하는 메타키인 Esc키를 누르고 이어서 x와 doctor를 입력해서 화면에 표시되는 전체적인 입력 형태가 ‘M-x doctor’가 되게 한 뒤에 Enter를 누르면 된다. 원하는 문구를 대화식으로 입력한 뒤에 매번 Enter를 두 번씩 누르면 그에 맞는 답변이 출력된다.
- 11 역자주\_dissociated-press 모드를 실행하면 문서의 내용이 단어와 문자를 기준으로 섞이기 때문에 원래의 형태는 없어지지만 재미있는 읽을거리가 생겨나는 이채로운 여흥이 된다. 이 모드로 들어가기 위해서는 메타키인 Esc키를 누르고 x와 dissociated-press를 차례대로 입력해서 전체적인 입력 형태가 ‘M-x dissociated-press’가 되게 한다. 이맥스는 명령행 완성 기능을 제공하기 때문에 dissociated-press를 모두 입력할 필요 없이 dissociate 정도의 단어만 입력한 뒤에 스페이스 키를 연이어 누르면 나머지 문자가자동으로 입력된다(이 책의 다른 글 ‘GNU 운영체제와 자유 소프트웨어 운동’ 중 ‘GNU 라이브러리 GPL’ 절의 리드라인 라이브러리 부분 참고).
- 12 역자주\_하노이 모드는 세 개의 기둥에 쌓아 올린 크기가 각각 다른 원형 판들을 큰 판이 작은 판 위로 오지 않도록 하는 규칙을 지키면서 하나의 기둥으로 모두 옮기는 하노이 탑 시뮬레이션이다. 하노이 탑은 재귀 호출 프로그램을 작성할 때 교과서와 같이 등장하는 것으로 Esc 키를 누른 뒤에 x와 hanoi를 입력해 하노이 모드로 들어갈 수 있다. 명령어 입력행에 ‘M-x hanoi’로 표시되면 된다.

- 13 역자주\_리처드 스톤먼은 원칙주의 자유 소프트웨어 운동가고 마이클 티만은 실용성을 추구하는 오픈 소스 사업이다. GNU 이맥스와 GDB, GCC 등은 모두 자유 소프트웨어라고 하는 것이 타당하지만, 티만은 이 글에서 자유 소프트웨어와 오픈 소스를 구분하지 않고 모두 오픈 소스라고 칭하고 있다. 시그너스 솔루션즈는 사업을 진행하면서 자유 소프트웨어라는 용어가 제품 판매 및 마케팅에 장애가 된다는 판단으로 ‘소스 코드를 이용할 수 있는 소프트웨어’라는 의미를 담은 소스웨어(sourceware)라는 용어를 만들어 사용하기도 했다. 오픈 소스라는 용어는 1998년이 되어서야 만들어진다. 이에 대해서는 이 책의 다른 글 ‘GNU 운영체제와 자유 소프트웨어 운동’ 중 ‘오픈 소스’ 절과 이 책의 또 다른 글 ‘오픈 소스에 대한 정의’ 부분을 참고할 수 있다. 이 용어가 만들어진 상세한 과정과 설명은 리처드 스톤먼의 전기『Free as in Freedom: Richard Stallman’s Crusade for Free Software, Sam Williams, O’Reilly, 2002』의 11장(<http://oreilly.com/openbook/freedom/ch11.html>)에서 참고할 수 있다.
- 14 역자주\_황금률(The Golden Rule)은 마태복음 7장 12절 「그러므로 무엇이든지 남에게 대접을 받고자 하는 대로 너희도 남을 대접하라. 이것이 율법이요 선지자니라」라는 문장의 뜻을 의미한다. 이 명제와 반대 방식의 서술(긍정형 대 부정형)이 논어 위령공(衛靈公) 편의 「내가 원치 아니하는 것을 남에게 베풀지 말라. 己所不欲(기소불욕)勿旅於人(물시여인)」인데 이를 은율(The Silver Rule)이라 한다. 같은 형식의 유대 랍비 힐렐의 말도 은율로 취급된다(이 책의 다른 글 ‘GNU 운영체제와 자유 소프트웨어 운동’ 중 <역자주 16> 참고).
- 15 역자주\_컴팩은 다시 HP로 인수·합병되었다(이 책의 다른 글 ‘GNU 운영체제와 자유 소프트웨어 운동’ 중 <역자주 3> 참고).
- 16 역자주\_썬 마이크로시스템즈(Stanford University Network Microsystems)는 2009년 4월 20일에 74억 달러(2009년 화폐가치 약 9조 4천억 원, 2013년 화폐가치 약 10조 원)의 금액으로 오라클(Oracle)에 인수·합병되었다.
- 17 역자주\_이 책의 다른 글 ‘GNU 운영체제와 자유 소프트웨어 운동’ 중 <역자주 8> 참고.
- 18 역자주\_원문에는 GCC 1.0판의 발표 날짜가 1987년 6월로 되어 있지만, 정확한 날짜는 1987년 5월 23일이다.
- 19 역자주\_여기서 회사로 옮긴 원문의 단어는 벤더(vendor)다. 벤더는 미국과 한국의 사업 형태가 일치하지 않는 점도 있고 단어 자체의 의미도 포괄적이기 때문에 상황에 따라 여러 행태로 번역할 수 있다. 라틴어 vendere에서 유래한 벤더는 ‘판매자(seller)’를 뜻하는데 유통 단계에 따라 생산자와 도매상, 소매상 모두가 벤더가 될 수 있다. 또한 외국의 소프트웨어를 들여와 국내에 공급하는 총판의 경우도 판매자, 즉 벤더가 된다. 사업의 규모와 선택에 따라 소프트웨어를 만드는 회사가 직접 생산과 유통, 공급, AS를 하면 이 회사가 해당 소프트웨어의 제조사인 동시에 벤더며, 유통과 공급을 다른 회사에 위탁하면 위탁받은 업체가 벤

더가 된다. 따라서 벤더는 문맥에 따라 생산업체, 제조업체, 공급업체, 배급업체, 유통업체, 총판, 재판매업체, 도매업체, 소매업체 등 여러 행태로 구체적으로 옮길 수 있다. 이 번역문은 단어가 가진 모호함을 피하기 위해 벤더로 직역하지 않고 문맥에 따라 다르게 옮겼다. 여기서는 VAX와 Sun-3 워크스테이션을 제작한 DEC와 썬 마이크로시스템즈가 컴퓨터와 함께 제공한 운영체제에 포함된 컴파일러를 말하고 있기 때문에 각각의 회사가 컴파일러의 벤더가 된다.

- 20 역자주\_MIPS는 ‘Million Instructions Per Second’의 약자로 1초 동안 실행할 수 있는 명령어의 수를 100만 개 단위로 나타낸 것이다. 이 절의 이야기 시점인 1987년은 중대형 컴퓨터 시대에서 PC 시대로의 전환이 본격화되는 때인데 당시의 PC용 CPU인 인텔 80386의 속도는 5MIPS였다. 따라서 중대형 컴퓨터 와 비슷하거나 더 나은 성능을 가지면서 개인이 구입할 수 있는 수준까지 내려간 가격을 고려하면 한계가 분명한 MS-DOS보다 PC에서 사용할 수 있는 유닉스 운영체제에 대한 수요가 생길 수밖에 없고 이것이 직·간접적으로 리눅스가 개발된 원인이 된다. 80386의 성능은 그 후 11.4MIPS까지 향상됐는데, 리눅스 토벌스가 1991년에 구입해 리눅스를 만들기 시작한 컴퓨터도 80386 시스템이다(당시의 PC 구입에 관한 이야기는 「리눅스 그날 재미로, 리눅스 토벌스 외, 한겨레신문사, 2001년」의 ‘나만의 터미널 에뮬레이션’ 부분에서 참고할 수 있다). 2013년 현재 PC용 CPU인 인텔 Core i7 4960X의 경우 대략 36,091MIPS의 속도를 갖고 있으며 HP의 최상위 유닉스 시스템 HP 인테그리티 슈퍼돔 2(Integrity Superdome 2)는 비슷한 성능의 인텔 아이테니엄(Itanium) 9500 시리즈 CPU를 32개까지 장착할 수 있다. (이 책의 많은 글에서 언급되는 PDP-10 시스템은 흔히 디지털로 부르는 DEC가 만든 것이며, 그 후 DEC를 인수·합병한 컴팩이 HP로 다시 인수·합병되었기 때문에 역사적인 이유에서 HP의 유닉스 시스템을 예로 들었다). 해커 공동체 시절 유닉스 시스템의 성능 기준점은 1MIPS였기 때문에 하나의 CPU를 기준으로 할 때 지난 30여 년간 3만 5천 배 정도의 성능 향상이 이루어진 것으로 볼 수 있다. 이 책의 다른 글 ‘GNU 운영체제와 자유 소프트웨어 운동’ 중 <[역자주 36](#)>에서 설명한 최대 메모리 용량과 관련해서 인텔 Core i7 시스템에서 사용할 수 있는 메모리는 CPU 모델과 메인보드, 운영체제 조합에 따라 다르지만, 현재 시판되는 PC를 기준으로 할 때 최대 64GiB이며 HP 인테그리티 슈퍼돔 2의 경우 8TiB의 메모리를 사용할 수 있다.
- 21 역자주\_시점이 조금 뒤이기는 하지만 오픈 소스 현상을 다룬 다큐멘터리 영화 「Revolution OS, J.T.S. Moore, 2001」에 이와 관련된 일화가 나온다. 다음은 영화 안에서의 티만의 말(<http://www.youtube.com/watch?v=CjaC8Pq9-V0>)이다. 「1990년에 모스크바 대학 학장이 저희를 방문했죠. 그 2주 전에 헬싱키에서도 그를 본 적이 있었어요. 어쨌거나, 그가 들렸고 리처드 스톤먼은 그에게 시그너스에 한번 가보라고 권했죠. 왜냐하면 그가 자유 소프트웨어 모델에 관심이 있었고 이해하려고 했거든요. 그게 어떻게 러시아의 모든 기업에 혁신을 주도록 자극할 수 있을지에 대해서 말이에요. 우린 그때만 해도 사업 계획을 외부에 밝히지 않았었죠. 왜냐하면 그게 잘 될 수 있을지 확신이 없었거든요. 실패하면 바보같이 보일까 두려웠죠. 그래도 저는 그에게 꽤 개방적이었어요. 말을 많이 할수록 그는 이렇게 고개를 흔들더군요. 그래서 결국 물

어봤죠. "뭐 잘못된 점이라도 있나요?" 그는 이렇게 말했죠. "이것은 러시아에서 성공하기에는 너무 공산주의적으로 들리는군요." (재생 지점 01:04:20~01:05:12)」인터뷰 중심으로 만들어진 이 다큐멘터리에는 이 책의 공저자 리처드 스톤먼, 리누스 토르발스, 에릭 레이먼드, 브루스 페렌스, 마이클 티만, 브라이언 벨렌도르프 등이 출연한다.

- 22 역자주\_아이작 뉴턴(Isaac Newton, 1642~1727). 중력과 운동 법칙을 연구해 뉴턴 역학이라 불리는 근대 물리학을 기초했다. 운동 제2법칙 ' $F=ma$ '는 흔히 '프린키피아'로 불리는 『자연철학의 수학적 원리(Philosophiae Naturalis Principia Mathematica)』에서 제시되었는데 <http://archive.org/details/philosophiaenat00newt>에서 이 책의 라틴어 원문을 참고할 수 있다.
- 23 역자주\_빅토르 위고(Victor Hugo, 1802~1885). 19세기 프랑스 낭만주의 작가로 『노트르담의 꼽추(Notre-Dame de Paris)』, 『레미제라블(Les Misérables)』등의 작품을 남겼다. 이 문구는 1877년에 출판된 『범죄의 역사(Histoire d'un Crime)』의 결론 부분에서 인용한 것이다(이 글에서 인용한 문구들은 원문의 내용을 그대로 적은 것이 아니라 마이클 티만이 자신의 표현으로 바꾼 것들이다). 이 책의 영문판은 <http://archive.org/details/cu31924028282519>에서 참고할 수 있다.
- 24 역자주\_이 맡은 요기 베라가 한 것이다. <역자주 52> 참고.
- 25 역자주\_애덤 스미스(Adam Smith, 1723~1790). 자본주의 경제 체제를 옹호하며 자유방임에 의한 자율적인 시장 경제를 강조했다. 이 문구는 그의 대표작 『국부론(An Inquiry into the Nature and Causes of the Wealth of Nations)』의 사상을 함축적으로 표현한 것이다. 이 책의 원문은 <http://archive.org/details/cu31924030402717>에서 참고할 수 있다.
- 26 역자주\_토마스 제퍼슨(Thomas Jefferson, 1743~1826). 「독립 선언서(Declaration of Independence)」를 기초한 미국의 제3대 대통령이다. 이 문구는 특허위원회(현재의 특허청장)으로 재임 중이던 1813년 8월 13일에 아이작 맥퍼슨(Isaac McPherson, 1770~1858)에게 보낸 편지 내용에서 인용한 것이다. 이 편지가 실린 저작선집은 <http://archive.org/details/writingsofthomas1314jeff>에서 참고할 수 있다.
- 27 역자주\_윌리엄 톰슨, 켈빈 남작 1세(William Thomson, 1st Baron Kelvin, 1824~1907). 영국의 물리학자로 열역학 분야에 지대한 공헌을 했다. SI 단위계의 국제 온도 표준을 나타내는 켈빈 온도 K가 그의 이름에서 유래한 것이며, OK인  $-273.15^{\circ}\text{C}$ 를 절대영도로 정의한다. 『Elements of Natural Philosophy, William Thomson, et al., Cambridge University Press, 1872』에 비슷한 취지의 설명은 보이지만 그의 저작 중에서 위 인용 문구의 정확한 출전은 찾을 수 없다. 이 문구의 원래 의미는 수식이나 개념을 설명하기 위한 수치나 도형(figure)에 잘못된 점이 있을 경우, 잘못된 부분을 찾아 더 깊이 이해할 수록 수식이나 개념이 더 가치 있게 된다는 것이다. 정확한 출전이 없기 때문에 이 글에서는 문맥에 맞게 문제로 의역했다.

- 28 역자주\_수백 명에서 수천 명의 사용자가 동일한 소프트웨어를 쓰는 대기업이나 대학 등을 대상으로 하는 이용허락 방식이다. 구매 수량이 많을수록 높은 할인율이 적용되며 보통 1년에서 3년 사이의 연간 계약으로 이루어진다. 회사 또는 학교 안과 같이 소프트웨어를 사용할 수 있는 장소와 범위를 지정하기 때문에 사이트 이용허락이라 불린다.
- 29 역자주\_토머스 에디슨(Thomas Edison, 1847~1931). 전등과 측음기 등을 발명한 20세기 최고의 발명가로 어린 시절에 달걀을 가슴에 품은 일화와 청각 장애를 딛고 일어선 불굴의 정신으로 유명하다. 그는 1,093 개의 기술 특허를 보유했으며 그중에 살상 무기와 관련된 발명이 1건도 없었다는 점을 스스로 자랑스럽게 생각했다고 한다. 그러나 무기는 아니지만 사형 집행용 전기의자는 그에 의해 발명되었다. 이 문구가 포함된 기사는 그가 사망한 뒤에 하퍼스 매거진(Harper's New Monthly Magazine) 1932년 9월호에 실렸는데, 기사의 원문(<http://harpers.org/archive/1932/09/edison-in-his-laboratory/>)에는 발명이 아닌 천재로 기록되어 있다.
- 30 역자주\_법은 형식에 따라 성문법(대륙법, 제정법, 로마법, 시민법)과 불문법(보통법, 코먼로, 판례법, 관습법)으로 구분한다. 의회가 제정하는 성문 절차를 거치지 않고도 기존의 판례(선례)가 구속력을 갖는 것이 보통법이며 영국과 미국의 법체계가 이에 해당한다.
- 31 역자주\_미국의 재판 사건 이름은 소송 당사자의 이름을 직접 사용해서 '원고(청구인) 대(對) 피고(피청구인)'와 같이 표시한다(한국은 법원이 자체 규칙에 따라 사건 번호를 부여한다). 연방제 국가인 미국은 주(State)마다 고유한 법이 있고 이에 대한 재판을 주 지방법원, 주 항소법원(한국의 고등법원), 주 대법원이 담당하지만 연방 정부 관할 사건은 전국에 별도로 설치된 연방 지방법원(미합중국 지방법원), 연방 항소법원(미합중국 항소법원), 연방 대법원(미합중국 대법원)이 맡는다. '로 대 웨이드' 사건은 원고인 제인 로가 임산부의 생명이 위험하거나 강간 또는 근친상간일 경우에만 낙태를 허용하던 텍사스주 낙태법이 개인의 자유와 권리 를 보장한 연방 헌법에 위반된다는 취지로 연방 지방법원에 제기한 것이며, 사건 이름에 사용된 웨이드는 주 정부를 대리한 검사의 이름이다. 이 사건은 로 이외에 다른 사람들이 함께 참여한 집단 소송이었고 정식 명칭은 'Jane Roe, et al. v. Henry Wade, District Attorney of Dallas County'다. 사건 이름에 사용하는 소송 당사자의 이름은 사생활 보호 등을 이유로 익명을 쓰기도 하는데 여성의 경우 제인 로(Jane Roe), 남성의 경우 존 도(John Doe)를 사용한다. 이 사건 제인 로의 실명은 노마 맥코비(Norma McCorvey)다. 연방 지방법원은 제인 로의 청구를 인용해 위헌을 인정했지만 낙태법의 파기를 명령하진 않았다. 또한 집단 소송인 중에 패소한 사람이 있고 당시 여성주의(페미니즘) 운동가들이 개입하는 등의 여러 이유로 인해 낙태에 대한 대법원 판단을 끌어내려는 목적으로 연방 대법원으로 (2심을 거치지 않고 바로 3심으로 가는) 비약상고 되었다. 이 시기에 낙태를 둘러싼 중요한 여러 사건이 연방 대법원에서 함께 심리되었다. 연방 대법원은 수정헌법 제14조(공민권)의 적법절차 조항을 근거로 당시 의학 수준에서 태아가 산모의 자궁 밖

에서도 생존할 수 있는 시기를 기준으로 삼아, 임신 후 28주 이내까지의 임신 중절 선택권을 인정했다(한국의 모자보건법도 동일한 판단 기준을 적용해 유전 질환이나 산모의 건강에 영향이 있는 예외적인 경우에 임신 중절을 허용했지만, 2009년 개정을 통해 현재의 의학 수준으로 태아가 생존 가능한 24주 이내로 기간을 변경했다). 이 사건의 판결은 1973년에 있었지만 여성의 낙태권 인정과 태아의 생명권 존중이라는 치열한 공방은 여전히 계속되고 있다. 소송 당사자 노마 맥코비는 재판이 진행되는 도중에 출산했으며 현재는 낙태 옹호(pro-choice)가 아닌 낙태 반대(pro-life)의 입장을 지지하고 있다(<http://www.youtube.com/watch?v=MYNy>NNq8Xg>). 이 사건에 대한 법리적인 내용은 판결문을 작성한 해리 블랙먼(Harry Blackmun, 1908~1999) 연방대법관의 전기 『블랙먼 판사가 되다, 린다 그린하우스, 청림출판, 2005년』의 제4장 '로 판결을 향해서' 부분에서 참고할 수 있다.

- 32 역자주\_법률과 오픈 소스를 기반으로 하는 서비스 사업 모델을 다음처럼 비교해 볼 수 있다.

사업 구분	법률 서비스 사업	오픈 소스 서비스 사업
회사의 형태	법무법인	아직 없음
핵심 인력	변호사	개발자
서비스 내용 및 노하우	소송 및 법률 실무 대리	오픈 소스 소프트웨어 개발 및 유지보수 대리
무료 이용 자산	판례	표준 및 오픈 소스 소프트웨어
무료 이용 자산 및 사업 경쟁력 증대 방법	사건 수임 및 승소	오픈 소스 소프트웨어 창작 및 표준 확립

- 33 역자주\_에스터 다이슨(Esther Dyson, 1951~). ICANN(Internet Corporation for Assigned Names and Numbers) 초대 의장이었던 그녀는 빌 게이츠(Bill Gates)와 함께 미국 컴퓨터 분야에서 가장 큰 영향력을 가진 사람 중 한 명으로 평가받는다. 1995년에는 시그너스 솔루션즈의 이사로 선임되기도 했다. 이 문구는 『Release 2.0: A Design for Living in the Digital Age, Esther Dyson, Broadway, 1997』의 마지막 장에서 인용한 것으로 이메일 서명에도 자주 사용한 그녀의 좌우명이다. 이 문구는 같은 실수를 반복하지 말라는 의미와 아무도 하지 않은 새로운 시도를 하라는 뜻이 있다. 한국에는 『인터넷, 디지털 문명이 열린다, 에스터 다이슨, 경향신문사, 1997년』이라는 제목으로 번역·출간되었다. 이 책의 전신은 그녀가 발간한 월간 뉴스레터 『Release 1.0』인데 지금은 폐간되었지만 오라일리 미디어가 과거 자료를 인터넷으로 무료 서비스(<http://radar.oreilly.com/r2/release1-0>)하고 있다.

- 34 역자주\_역자주\_한국의 '나 홀로 시리즈'처럼 개인이 변호사 없이 직접 법률 실무를 할 때 참고할 수 있는 책을 전문으로 내는 곳이다.
- 35 역자주\_2013년 현재 캘리포니아 지역의 시간당 법률자문 평균 비용은 200~300달러 수준이며, 한국의 경우 시간 자문일 경우 30~60만 원 수준이다.

- 36 역자주\_존 길모어(John Gilmore, 1955~). 썬 마이크로시스템즈 창업에 참여해 이미 백만장자가 된 후에 GNU 프로젝트에서 활동하다가 마이클 티만, 데이비드 헨켈-윌리스와 함께 시그너스 솔루션즈를 공동 창업했다. 이 문구는 시그너스 솔루션즈가 처음 만든 회사 티셔츠에 넣은 슬로건이었다. 1995년에 사업에서 은퇴한 후에 모아놓은 재산을 기반으로 인터넷상의 자유와 공유를 위한 여러 활동을 하고 있다(은퇴 후에도 시그너스의 주식은 갖고 있었다). 블루 리본 캠페인으로 유명한 전자개척재단(<http://www.eff.org>)을 공동 설립했으며 뉴스그룹의 alt.\* 계층을 만들었고 2009년 제12회 자유 소프트웨어 재단상을 수상하기도 했다. 그가 쓴 시그너스 솔루션즈의 마케팅 기록(<http://www.toad.com/gnu/cygnus/>)은 이 글과 관련해 참고할 만한 좋은 자료다.
- 37 역자주\_손자(孫子, Sun Tzu, BC.500). 이 문구는 중국 춘추전국시대의 사상가 손자의 『병법(孫子兵法, The Art of War)』 군형편(軍形篇)에서 인용한 것으로 전문의 내용과 뜻은 다음과 같다. 「孫子曰(손자왈)昔之善戰者(석지선전자) 先爲不可勝(선위불가승)以待適之可勝(이대적지가승). 손자가 말하기를, 예전에 용병을 잘하는 자는 먼저 적군이 아군을 이길 수 없도록 완벽한 태세를 갖춘 뒤에 아군이 적군을 이길 수 있을 때를 기다렸다.」 손자병법에서 가장 널리 알려진 자락인 ‘나를 알고 적을 알면 백 번 싸워도 절대 위태롭지 않다’는 뜻의 「知彼知己(지피지기) 百戰不殆(백전불태)」는 모공편(謀攻篇)에 실려 있다. 이 글의 내용과 관련해 참고할 만한 책으로 손자병법과 경영학 이론을 연결해 설명한 『디지털 손자병법, 현정석, KT문화재단, 2004년』([http://www.cefy.org//lib/download.php?file\\_name=art\\_of\\_war.pdf&save\\_file=a\\_201201251523370.pdf](http://www.cefy.org//lib/download.php?file_name=art_of_war.pdf&save_file=a_201201251523370.pdf))이 있다.
- 38 역자주\_1989년에는 아직 자유 소프트웨어 커널이 존재하지 않았기 때문에 시그너스 솔루션즈의 초기 계획에 커널 개발이 포함되어 있었다. 리눅스 0.01판은 1991년 9월 17일에 발표되었고 1.0판은 1994년 3월 14일에 발표되었다(이 책의 다른 글 ‘GNU 운영체제와 자유 소프트웨어 운동’ 중 ‘GNU 허드’ 이후 부분 참고).
- 39 역자주\_상품 박스를 비닐로 수축 포장한 제품을 말한다. 슈링크랩은 일반적으로 제품 포장이 아닌 이용허락 방식을 말하는 것으로 비닐 포장을 개봉하는 행위가 이용허락 계약에 동의하는 의사표시로 간주된다. 슈링크랩에서 파생된 이용허락 방식으로 클릭랩(click-wrap)이 있다. 이는 팝업창 등을 통해 나타나는 동의 화면에 클릭하는 방식으로 이용허락이 이루어지는 것이다. 슈링크랩 제품을 흔히 박스 제품이라고 표현하기도 한다.
- 40 역자주\_〈역자주 19〉에서 설명한 용어 ‘벤더’와 연관 지어 보면, 서드파티 업체는 독립 소프트웨어 공급업체 또는 독립 소프트웨어 벤더(ISV: Independent Software Vendor)로 바꿔 사용할 수 있다.
- 41 역자주\_EDS(Electronic Data Systems)는 1962년에 설립된 세계 최대의 아웃소싱(outsourcing) 전문 업체다. 창업자 로스 페로(Ross Perot)는 IBM(International Business Machines)의 영업사원이었으

나 당시 IBM 7070 시스템의 남는 용량과 성능을 컴퓨터를 구입할 여력이 안 되는 기업에 임대하는 사업을 제안했다가 거절당하자 IBM을 나와 최초의 IT 서비스 업체 EDS를 창업했다. 한국에는 1987년에 현재의 LG(Lucky Goldstar)와 합자·설립한 LG-EDS가 대법원 등기업부 전산화 등을 포함한 다양한 분야에서 아웃소싱 사업을 펼쳤다. 2001년에 LG가 지분 전체를 매입한 후 2002년 LG-CNS(Consulting and Solutions)로 사명을 변경했으며, 이와 별도로 1996년에 설립된 EDS코리아와 2007년 대우정보시스템 DIS(Daewoo Information Systems)가 합자·설립한 DIS-EDS가 있다. 2008년에 HP가 EDS를 인수·합병하면서 한국에서도 2009년 한국HP가 EDS코리아를 인수·합병했다. EDS와 DIS-EDS의 현재 이름은 각각 HP 엔터프라이즈서비스(HP Enterprise Services)와 HP-DIS다.

- 42 역자주\_이후에 나올 RTOS와 임베디드 시스템의 개념을 여기서 함께 설명한다. 실시간 운영체제 RTOS(Real Time Operating System)는 운영체제의 실시간 처리 기능을 강조한 개념이다. 따라서 요청된 작업에 대한 처리를 시간 지연 없이 바로 수행할 수 있는 리눅스도 RTOS가 될 수 있다. RTOS를 이해하기 위해서는 먼저 임베디드 시스템의 개념을 이해하는 것이 도움이 된다. 컴퓨터 이외에 스마트폰이나 전자 수첩, 통신 기기, 세탁기, 전기밥통, 셋톱 박스 등과 같은 가전제품 안에는, 리눅스나 윈도우 XP, 윈도우 7과 같은 모든 기능을 갖춘 범용 운영체제는 포함되어 있지 않지만 자동으로 작동하는 자체적인 운영 환경을 유지하기 위해 소규모 운영체제나 애플리케이션을 마이크로프로세서나 메모리 칩 안에 내장하는데 이러한 형태의 제품을 임베디드 시스템(embedded system) 또는 내장형 시스템 또는 독립 장비라는 말로 표현한다. 임베디드 시스템에 내장되거나 애플리케이션을 개발할 수 있는 환경을 제공하는 것이 RTOS이며 임베디드 시스템용 애플리케이션을 개발하기 위해서는 호스트(host)와 타깃(target)이라는 2개의 작업 환경을 사용하게 된다. 네트워크 연결을 통해 인터넷 검색이 가능한 새로운 모델의 전자수첩을 개발하는 경우를 예로 들면, 이 또한 전원을 이용해서 자체적으로 작동하는 제어 프로그램이 사용되어야 하므로 하나의 임베디드 시스템이라고 할 수 있다. 그러나 전자수첩 안에는 전용 웹 브라우저를 개발하는 데 필요한 하드디스크나 편집기, 컴파일러, 디버거 등의 개발 도구가 없기 때문에 개발 환경을 제공할 수 있는 좀 더 큰 규모의 시스템이 필요한데, 이러한 용도로 사용하는 개발 모체를 호스트라는 말로 표현하고 호스트에서 만들어진 프로그램이 실제로 실행되는 최종 플랫폼을 타깃이라고 한다. 임베디드 시스템을 개발할 때는 호스트와 타깃을 별개의 아키텍처로 선택할 수 있기 때문에 애플리케이션의 개발은 인텔 x86 아키텍처에서 이루어지지만 애플리케이션이 실행될 실제 타깃은 모토로라의 PowerPC(Performance Optimization With Enhanced Risc, Performance Computing)나 ARM(Advanced Risc Machines)으로 선택하는 등의 이질적인 아키텍처 조합이 가능하다. 본문의 뒷부분에 가면 GNUPro 툴킷이 175개 조합의 호스트/타깃 플랫폼을 지원할 수 있다는 언급이 나오는데 이때의 호스트/타깃은 바로 이러한 의미며 호스트와 타깃이 동일한 아키텍처 일 경우에는 네이티브(native)라는 용어를 사용한다. 따라서 인텔 x86 프로세서가 장착된 호스트 컴퓨터에 리눅스를 설치한 뒤에 GCC 컴파일러로 네이티브 코드를 생성했다면 이때 생성된 실행 파일은 컴파일 호스

- 트와 동일한 타깃 플랫폼인 인텔 x86 시스템에서만 실행할 수 있는 코드가 된다. 임베디드 시스템을 만들기 위해서는 특정한 호스트에서 컴파일된 실행 파일이 다른 아키텍처의 타깃에서 실행될 수 있어야 하는데, 이러한 형태로 실행 파일을 만들어 줄 수 있는 컴파일러를 크로스 컴파일러(cross compiler) 또는 교차 컴파일러라고 한다. 크로스 컴파일러는 임베디드 시스템 개발에만 국한되지 않고 이미 존재하는 특정한 프로그램을 다른 아키텍처 환경으로 이식할 때도 널리 사용된다. 크로스 컴파일러와 대비해서 호스트와 동일한 아키텍처에서만 작동하는 실행 파일을 만드는 컴파일러를 네이티브 컴파일러라고 한다. 인텔 x86용 네이티브 컴파일러로 만든 실행 파일은 x86 시스템에서만 사용할 수 있다. 만약 호스트와 타깃이 다른 아키텍처라면 호스트에서 크로스 컴파일러로 만든 실행 파일은 호스트에서 만든 것임에도 불구하고 타깃에서만 실행될 뿐 호스트에서는 당연히 실행되지 않는다. 크로스 컴파일러는 --host와 --target 옵션을 사용해서 GCC 소스 코드를 컴파일하는 방법으로 만들 수 있으며 <http://web.archive.org/web/20000819000833/>, <http://www.objsw.com/CrossGCC/>, <http://sourceware.org/ml/crossgcc/>, <https://www.kernel.org/pub/tools/crosstool/>를 통해 관련 정보를 참고할 수 있다. 본문의 뒷부분에 등장하는 eCos는 임베디드 시스템 개발을 위해 시그너스 솔루션즈가 만든 RTOS의 이름이다.
- 43 역자주\_SPARC(Scalable Processor ARChitecture)은 썬 마이크로시스템즈가 1985년에 개발한 RISC 마이크로프로세서의 이름이다. SPARC 컴퓨터에 탑재하는 썬의 기본 유닉스 운영체제는 SunOS와 솔라리스(Solaris)다.
- 44 역자주\_인스트럭션 스케줄링(instruction scheduling)은 기계어 명령(instruction)을 재배치하는 등의 방법으로 실행 파일의 성능을 높이기 위해 컴파일러가 수행하는 최적화 기능이며, 이러한 기능을 담당하는 부분이 인스트럭션 스케줄러다. GCC가 썬의 컴파일러보다 20% 정도 느리다는 말의 의미는 컴파일 결과로 만들어진 프로그램의 실행 속도가 느리다는 것을 말한다. 이 글의 <[역자주 20](#)>의 본문에서 관련 내용을 참고할 수 있다. <[역자주 19](#)>의 본문에서 언급한 Sun-3 시스템에는 CISC 칩인 모토로라 68020이 사용되었다.
- 45 역자주\_또 다른 코드 최적화 방법의 하나다. 이 책의 다른 글 ‘소프트웨어 공학’ 중 <[역자주 10](#)>과 본문에서 관련 내용을 참고할 수 있다.
- 46 역자주\_이 책의 다른 글 ‘GNU 운영체제와 자유 소프트웨어 운동’ 중 ‘첫 번째 단계’ 절에서 언급한 것처럼 GCC 컴파일러는 CISC 아키텍처인 모토로라 68000 시스템(m68k)에 맞춰 개발된 것이었기 때문에 RISC 아키텍처로 최적화해 이식하기 위해서는 새로운 기능들이 추가되어야 했다. 컴퓨터는 CPU의 마이크로 명령 구성 방식에 따라 CISC(Complex Instruction Set Computer)와 RISC(Reduced Instruction Set Computer) 두 가지로 크게 구분한다. C나 포트란과 같은 고급 프로그래밍 언어로 작성된 프로그램은 컴파일 과정을 거쳐 CPU에서 직접 실행될 수 있는 기계어로 번역되는데, 만약 고급 언어의 특정 기능을 수행

할 수 있는 명령어를 CPU 수준에서 직접 지원하게 되면 컴파일 과정이 단순해지고 시스템의 전체적인 수행 능력을 향상할 수 있다. 이러한 장점을 살리기 위해 고급 언어의 기능을 수행할 수 있는 명령어를 가능한 한 많이 포함해 설계한 시스템이 CISC다. 그러나 명령어의 수가 많아지면 CPU 내부 회로와 명령어 구성 방식이 복잡해져서 실행 속도가 늦어지는 단점이 있다. RISC 아키텍처가 고안된 가장 큰 이유는 명령어의 수를 가능한 최소로 줄이고 모든 연산이 CPU 안에서만 수행될 수 있도록 해 실행 속도를 높이기 위한 것이다. CISC와 RISC 아키텍처는 컴퓨터의 수행 능력과 컴파일 과정을 최적화시키려는 목적으로, 실행 속도를 최대화하려는 두 가지 다른 목적에 따라 만들어진 것으로 시스템 설계의 측면에서 볼 때 하나의 시스템 안에 모든 기능을 최적화시켜 구현할 수는 없기 때문에 하나를 충족시키면 다른 하나를 희생할 수밖에 없는 양자택일, 즉 트레이드오프(tradeoff)를 해야 한다. 대표적인 CISC 아키텍처로는 DEC의 VAX와 IBM 370, 인텔의 80x86이 있다. 시그너스 솔루션즈가 GCC 상용 패키지를 본격적으로 개발하기 시작한 1989년은 인텔이 1985년과 1988년에 각각 발표한 386DX와 386SX에 이어 486 프로세서를 발표함으로써 32비트 CISC 아키텍처에 의한 개인용 컴퓨터의 전성기가 시작되던 시기에 해당한다. 또한 이 시점은 리눅스의 모태가 된 PC용 유닉스 운영체계 미닉스(Minix)가 인터넷을 통해 전 세계적으로 알려지던 때였으며, 17가지 이상의 컴파일러가 난립하던 상업용 PC 컴파일러 시장을 분당 7천 행이라는 당시로써는 경이로운 속도로 컴파일할 수 있는 볼랜드(Borland)의 Turbo C/C++ 컴파일러가 석권하던 때이기도 하다. 그러나 호환성이라는 고육지책으로 인해 다른 아키텍처로 돌아설 수 없을 만큼 평창해 버린 개인용 PC 이외의 시장에서는 실행 속도를 극대화하기 위해 CISC에서 RISC로 아키텍처의 주력이 변경되었는데, 1984년 이후부터 새롭게 등장한 MIPS와 SPARC, PA-RISC, PowerPC, Alpha, ARM 등의 프로세서는 모두 RISC 아키텍처에 해당하며 1995년부터 본격적으로 판매된 인텔 펜티엄 프로세서에도 RISC 기술이 부분적으로 도입되었다.

- 47 역자주\_C++는 벨 연구소(AT&T Bell Laboratories)의 비아르네 스트로우스트룹(Bjarne Stroustrup)이 1983년에 발표한 언어다. C++ 언어의 표준은 IETF가 아닌 ISO/IEC에서 이루어지며 최근의 표준은 2011년에 승인된 C++ 11(<http://www.open-std.org/jtc1/sc22/wg21/>)이다. 오라일리 미디어가 출판한 『Masterminds of Programming: Conversations with the Creators of Major Programming Languages, Federico Bianuzzi, et al., O'Reilly Media, 2009』 중에서 스트로우스트룹과의 인터뷰 내용은 [http://www.stroustrup.com/masterminds\\_chapter\\_1.pdf](http://www.stroustrup.com/masterminds_chapter_1.pdf)에서 참고할 수 있다.
- 48 역자주\_GNU C++ 컴파일러는 C 언어 컴파일러인 GCC에 C++ 확장 패키지를 추가한 것이다. GNU C++ 컴파일러는 GNU C++ 또는 G++라고 부르며 컴파일에 사용하는 명령어 또한 c++와 g++ 두 가지 형태를 모두 사용할 수 있다.
- 49 역자주\_완비된 개발 환경 및 이를 지원하는 도구 모음을 말할 때 툴킷(toolkit), 툴셋(toolset), 툴체인(toolchain), 툴박스(toolbox), 툴스위트(toolsuite), 스위트(suite), 프레임워크(framework) 등의 용어

를 거의 같은 뜻으로 사용하지만 이 글에서는 원문에 사용된 용어를 각각 그대로 옮겼다. 특히 툴체인의 경우 chain이란 단어의 의미처럼 연쇄적으로 사용되는 도구들의 모음을 말하기 때문에 자유 소프트웨어 개발 프레임워크인 GNUPro 툴킷의 성격을 잘 표현한 용어라고 볼 수 있다. 자유 소프트웨어 개발 툴셋인 GNUPro는 가장 인기 있는 임베디드 시스템 개발용 프레임워크다.

- 50 역자주\_GAS(Gnu ASsembler)와 GLD(Gnu LoaDer)는 GNU 어셈블러와 링커의 이름이며 실행 파일 명은 각각 as와 ld이다. GAS와 GLD의 한국어 매뉴얼은 <http://korea.gnu.org/manual/release/as/>, <http://korea.gnu.org/manual/release/ld/>에서 참고할 수 있다.
- 51 역자주\_GNU C 라이브러리 glibc 1.0판은 1992년 2월 18일에 발표됐다.
- 52 역자주\_요기 베라(Yogi Berra, 1925~). 미국 메이저리그의 전설적인 야구 선수로 여러 분야에서 인용되는 많은 말을 남겼다. 요기 베라의 말로 잘못 알려진 것이 많아 그 자신도 “내가 한 모든 말이 정말 내가 한 것은 아니다”라고 언급하기도 했는데, 이 문구는 요기 베라가 아닌 미국의 애니메이터 월트 켈리(Walt Kelly, 1913~1973)가 그린 만화 주인공 포고(Pogo)가 한 말을 잘못 인용한 것이다. 요기즘(Yogi-isms)이라고도 부르는 그의 어록은 『The Yogi Book, Yogi Berra, Workman Publishing, 1998』으로 출판되었고 한국에는 『끝날 때까지는 끝난 게 아니다, 요기 베라, 시유사, 2001년』으로 번역·출판되었다.
- 53 역자주\_여기서 말하는 GNU 제품은 GNUPro가 출시되기 이전에 ‘시그너스 개발자 키트(Cygnus Developer's Kit)’ 또는 ‘CDK GNU 솔루션’이라는 이름으로 판매하던 상품을 말한다.
- 54 역자주\_커베로스는 SSH(Secure SHell)가 보편화 되기 이전부터 사용되던 네트워크 보안 인증 프로토콜의 한 종류이며, 시그너스가 수정 및 개발을 통해 판매하던 상품의 이름은 KerbNet이다. 커베로스와 SSH, PGP(Pretty Good Privacy) 등의 암호화 도구는 관련 기술을 전략 무기로 간주하는 미국 연방법에 따라 외국으로의 수출이 철저히 규제되었지만, 전자상거래의 보안 신뢰도를 높이기 위해 윈도우 2000에 암호화 기술을 내장하려고 했던 마이크로소프트와 업계의 로비 때문에 관련법이 완화되었다. 그 이전에 PGP 등의 암호화 프로그램을 한국에서도 사용할 수 있었던 이유는 특허나 특정 국가의 법률에 따라 규제되던 기술이더라도 일단 해당 기술의 내용이 일반적으로 널리 알려지게 되면 별다른 제한 없이 사용할 수 있다는 국제 저작권법상의 해석에 따라 미국 밖으로 불법 유출된 프로그램 소스 코드를 이용해 국제 판으로 제작된 다른 프로그램을 사용할 수 있었기 때문이다. 이 책의 다른 글 ‘GNU 운영체제와 자유 소프트웨어 운동’ 중 ‘가려운 곳을 짚는다?’ 절에 나오는 GnuPG는 PGP를 대체하기 위해 개발된 것이다. 커베로스에 대한 자세한 사항은 <http://web.mit.edu/kerberos/>에서 참고할 수 있다.
- 55 역자주\_이 소프트웨어는 GNATS(GNats: A Tracking System, 그네츠)를 말한다. GNATS는 Ada(에이다) 언어 컴파일러인 GNAT(Gnu NYU Ada Translator, 그네트)와는 다른 것이다(<http://www.gnu.org>).

[org/software/gnats/](http://org/software/gnats/)). 펑귄을 리눅스 마스코트로 쓰는 것처럼 유닉스 계열의 프로그램은 전통적으로 동물을 마스코트나 로고로 많이 사용한다. 이 때문에 이름을 만들 때 마스코트로 삼을 동물을 함께 고려하기도 하는데, gnat(내트)는 최근 한 드라마를 통해 '각다위'로 알려진 '각다귀'를 뜻하는 단어이기 때문에 각다귀가 GNATS의 로고로 사용되기도 한다. 버그(벌레) 추적 시스템의 상징으로 벌레가 사용된 셈이다.

- 56 역자주\_모든 컴퓨터 파일은 컴퓨터가 이해할 수 있는 0과 1의 바이너리 데이터로 저장되기 때문에 이를 바이너리 파일 또는 이진 파일이라고 부른다. 텍스트 파일 TXT나 오디오 파일 포맷 MP3, 이미지 포맷 JPEG 등은 용도에 따른 구분이며 컴퓨터에 저장되는 형태는 모두 동일한 바이너리 포맷이다. 그러나 특정 프로그램이 바이너리 데이터를 처리하는 구현 방식에는 차이가 있을 수 있다. 텍스트 또는 아스키(ASCII: American Standard Code for Information Interchange) 파일이라고 부르는 것은 사람이 이해할 수 있는 영문 알파벳을 컴퓨터 화면에 직접 표시할 수 있는 (인코딩 및 디코딩) 처리 방식이며, 이 관점을 기준으로 하면 컴퓨터 파일은 텍스트 파일과 바이너리 파일 두 가지로 구분할 수 있다. 즉 모든 컴퓨터 파일은 바이너리 파일이지만 사람이 직접 이해할 수 있는가를 기준으로 텍스트 파일과 바이너리 파일 두 가지로 크게 구분하며, 바이너리 파일은 용도나 구현 방식에 따라 다시 수많은 이름으로 세분된다. 실행 파일이나 오브젝트 파일도 텍스트 파일이 아니기 때문에 모두 바이너리 파일이다. 129쪽 표의 개발 도구 중에서 컴파일러와 어셈블러는 텍스트 편집기 등을 이용해서 사람이 이해할 수 있는 프로그래밍 언어로 만든 소스 코드를 읽어 들이기 때문에 다른 도구들과 달리 입력 포맷이 아스키, 즉 텍스트로 되어 있다. 영문 알파벳을 기준으로 하는 아스키와 달리 현재의 프로그램 소스 코드는 한글이나 한자 등의 비영어권 언어를 모두 포함할 수 있기 때문에 큰 틀에서 아스키라 불릴 수는 있지만, 정확하게 말하면 유니코드인 UTF-8(Unicode Transformation Format 8) 파일이며 GCC의 현재 기본 입력 출력 포맷은 아스키가 아닌 UTF-8로 바뀐 상태다. GCC가 소스 코드를 읽어 컴파일한 파일은 기본적으로 바이너리 파일이지만 디버깅이나 바이너리 유ти리티를 이용한 작업에 사용할 수 있는 여러 가지 유용한 정보가 삽입되는데 이러한 정보(심벌)은 nm이나 objdump, readelf 등의 명령어를 이용해서 사람이 읽을 수 있는 문자 정보 형태로 출력할 수 있다. 이것이 가능한 이유는 GCC가 생성한 바이너리 파일이 아스키 또는 UTF-8로 인코딩된 부분을 포함하고 있기 때문이다. 또한 GCC는 오브젝트 파일이나 실행 파일뿐 아니라 어셈블러 소스와 전처리 파일도 생성할 수 있기 때문에 이러한 측면에서 129쪽 표의 GCC 출력 포맷이 아스키로 표시되어 있다. 그러나 출력 파일이 아스키 인코딩 부분을 포함한다고 하더라도 여기에는 GCC가 만든 바이너리 코드가 함께 섞여 일반 텍스트 파일과 다른 형태로 저장되기 때문에 텍스트 편집기로 열어도 일반 문서처럼 읽을 수 있는 것은 아니다. 129쪽 표에 있는 개발 도구들은 컴파일러와 디버거를 제외하면 모두 binutils 패키지에 포함되어 있으며 실행 파일의 이름은 차례대로 as(어셈블러), ar(아카이브), ld(링커), size, strip, nm이다. GNU C 컴파일러의 경우, 이 글을 쓸 당시에는 시그너스 솔루션즈에 의해 GCC에서 갈라져 나온 EGCS(Experimental GNU Compiler System, 에그스)가 별개의 프로젝트로 진행되었으나 1999년에 GCC로 다시 통합되었다. binutils 패키지에 대해서는

<http://www.gnu.org/software/binutils/>에서 자세한 정보를 참고할 수 있다.

- 57 역자주\_컴파일러와 어셈블러가 생성하는 오브젝트 파일과 실행 파일 포맷은 최초의 유닉스에서 사용하던 a.out(Assembler.OUTPUT)부터 다양하게 발전했다. 리눅스는 유닉스 시스템V 릴리스4부터 도입된 elf(Executable and Linkable Format)를 기본 포맷으로 사용한다. ecoff와 xcoff는 유닉스 시스템V 릴리스3에서 사용된 coff(Common Object File Format)의 확장 포맷이며, ieee695는 IEEE에서 표준화한 'IEEE Standard for Microprocessor Universal Format for Object Modules' 포맷이다.
- 58 역자주\_BFD와 관련한 재미있는 일화가 있다. 2010년 3월 23일 미국 건강보험 개혁법안(Health Care Reform) 서명식에서 조 바이든(Joe Biden) 부통령이 버락 오바마(Barack Obama) 대통령에게 법안 통과를 축하하며 "This is a Big Fucking Deal"이라고 귀속말로 얘기한 것이 마이크로 전달돼 큰 논란이 되었는데, 그 후 오바마 대통령은 부통령에게 백악관 출입기자단 만찬 참석을 공개적으로 권유하며 이 표현을 "This is a Big Fucking Meal"로 재치 있게 인용했다.
- 59 역자주\_GNUPro의 릴리스는 분기마다 이루어졌다. 130쪽 표의 릴리스 이름은 1995년부터 95q1, 95q2, 95q3, 95q4, 96q1 등의 형태로 바뀌었다. 소프트웨어 신판이나 개정판을 발표할 때 흔히 기준 판과의 구별을 위해 번호를 사용하는 데 이때 사용하는 용어로 버전(version)과 릴리스(release), 코드네임(codename) 등이 있다. 릴리스는 버전보다 작은 수준의 변경이나 현재 판매하고 있는 출시 버전을 의미하기도 하지만 소프트웨어 회사의 선택에 따라 별다른 구분 없이 혼용되기도 한다. 전자의 기준을 따를 경우 'GNUPro version 3, release 4'는 'GNUPro 3판 4쇄'와 같은 형식으로 번역할 수 있지만 이 글은 릴리스와 버전을 큰 구분 없이 사용하고 있기 때문에 version과 release를 각각 '판'과 '릴리스'로 번역했다. 이 글은 유닉스 시스템V와 GNUPro 두 가지에 대해 버전 대신 릴리스란 용어를 사용한다. 코드네임은 여려 이유로 숫자 대신 사용하거나 함께 병기하는 일종의 별칭이다. 예를 들어 데비안 GNU/리눅스 배포판의 5.0판, 6.0판, 7.0판의 코드네임은 각각 lenny, squeeze, wheezy이다.
- 60 역자주\_GNUPro 툴킷은 리눅스를 포함한 유닉스용과 MS 윈도우용이 모두 개발되었으며, <http://web.archive.org/web/19970330040733/www.cygnus.com/pubs/gnupro/>에서 자세한 정보를 참고할 수 있다(레드햇과의 인수·합병 이후에는 주소가 <http://www.redhat.com/services/custom/gnupro/>로 변경되었다).
- 61 역자주\_애플라이드 머티어리얼즈는 캘리포니아주 산타 클라라(Santa Clara)에 위치한 반도체 장비 및 서비스 회사다.

- 62 역자주\_매출액의 연도별 원화 환산 가치는 해당 연도와 2013년을 기준으로 각각 다음과 같다.

연도	매출액	해당 연도 환폐가치	2013년 환폐가치
1990년	725,000달러	약 5억 1천만 원	약 12억 2천만 원
1991년	1,500,000달러	약 11억 원	약 24억 원
1992년	2,800,000달러	약 22억 원	약 45억 7천만 원
1993년	4,800,000달러	약 38억 5천만 원	약 76억 원
1994년	5,700,000달러	약 45억 8천만 원	약 85억 원

- 63 역자주\_연평균성장을 CAGR(Compound Annual Growth Rate)은 사업 초기부터의 성장을 연도별로 나타낼 수 있는 지표이며 다음 공식을 통해 산출한다.

$$\left[ \left\{ \left( \frac{\text{마지막 연도 총액}}{\text{시작 연도 총액}} \right)^{\left( \frac{1}{연도 수} \right)} \right\} - 1 \right] \times 100\%$$

1991년의 CAGR은 시작 연도인 1990년과 1991년의 수치를 사용하고, 1993년의 CAGR은 1990년과의 1993년의 수치를 사용해서 각각 다음과 같이 계산한다.

$$\left[ \left\{ \left( \frac{1500000}{725000} \right)^1 \right\} - 1 \right] \times 100 = 106\%$$

$$\left[ \left\{ \left( \frac{4800000}{725000} \right)^{\left( \frac{1}{2} \right)} \right\} - 1 \right] \times 100 = 87\%$$

- 64 역자주\_알렉산더 그레이엄 벨(Alexander Graham Bell, 1847~1922). 이 문구는 1876년 3월 10일에 벨과 그의 조수 웨슬리(Thomas Watson) 간에 있었던 세계 최초의 전화 통화 내용이다. 이 통화는 인간의 음성을 전파로 이동시킬 수 없다는 고정 관념을 깨고 새로운 커뮤니케이션 문명을 연 역사적인 사건이지만 벨이 전화를 발명하게 된 최초의 동기는 청각 장애를 가진 그의 아내 메이블 허버드(Mabel Hubbard)를 위한 애틋한 사랑에서였다.
- 65 역자주\_각각의 소프트웨어에 대한 자세한 정보는 <http://www.gnu.org/software>의 개별 프로젝트를 통해 참고할 수 있다.
- 66 역자주\_회사 A가 돈을 들여 만든 결과물이지만 오픈 소스이기 때문에 누구나 자유롭게 그리고 공짜로 사용 할 수 있게 된다. 경쟁업체도 그 '누구나'에 포함되기 때문에 특정 분야에서 사업을 하는 회사 A의 투자는 과적으로는 그 분야 경쟁자에게 좋은 일을 시키는 꽂이 된다는 문제가 제기될 수 있다.
- 67 역자주\_레드햇으로 인수·합병되어 소멸할 때까지 시그너스는 비공개회사였다. <[역자주 3](#)> 참고.

- 68 역자주\_제프리 무어가 그의 저서 『Crossing the Chasm, Geoffrey Moore, Harper Business, 1991』에서 제시한 개념을 말한다. ‘틈’ 또는 ‘협곡’을 의미하는 단어 캐즘은 첨단 기술 관련 신생 기업이 성공할 수 있는 초기 시장과 대기업이 선점한 주류 시장 사이의 벽을 말하며 제프리 무어는 이러한 장벽을 극복하는 방법을 마케팅 차원에서 설명한다. 한국에는 『벤처 마케팅, 세종서적, 1997년』으로 번역판이 출판되었고 1999년 영문 전면 개정판은 『캐즘 마케팅, 세종서적, 2002년』으로 출판되었다. 썬과 오라클을 포함한 많은 IT 기업의 성공 사례가 분석되어 있지만 오픈 소스 관련 기업은 포함되어 있지 않다.
- 69 역자주\_격주간 경제 전문지 포천(Fortune magazine)이 선정하는 500대 기업 중 상위 100개 기업을 말한다. 2005년 이전의 포천 500은 [http://money.cnn.com/magazines/fortune/fortune500\\_archive/](http://money.cnn.com/magazines/fortune/fortune500_archive/)에서, 2005년 이후의 포천 500은 <http://money.cnn.com/magazines/fortune/fortune500/>에서 참고할 수 있다.
- 70 역자주\_시스너스 솔루션즈의 조직 구성은 (1) 이사회, (2) 대표이사 사장, 기술, 영업마케팅, 재무 담당 부사장 3명으로 구성된 경영관리팀(경영진), (3) 기술팀, 영업마케팅팀, 고객지원팀 등이다. 작고 유기적인 벤처 기업의 특성상 초기에는 부서의 구분이 모호하고 뒤로 갈수록 조직의 형태와 이름이 여러 가지로 바뀌지만 이 번역문에서는 통일성을 위해 시그너스 솔루션즈의 부서를 직접 언급해야 할 경우, 부서의 이름은 위의 기준을 사용하고 기술팀 등의 부문별 매니저는 관리직원으로, 기술팀 엔지니어는 기술직원으로, 경영관리팀은 상황에 따라 경영진과 경영관리팀으로 옮겼다. 경영관리팀은 최고 임원들 외에 일반 관리직원을 함께 포함한 부서 조직을 언급할 때 사용했다.
- 71 역자주\_미국 세법이 인정하는 기업회계 재고 자산 평가 방법의 하나로 금액의 증감을 중심으로 산정한다. 그러나 한국이 2011년(미국의 경우 2014년)부터 도입한 국제회계기준 IFRS(International Financial Reporting Standards)에서는 재고 자산 처리 방법으로 인정되지 않는다.
- 72 역자주\_OEM의 원래 뜻은 'Original Equipment Manufacturer'지만 원문에서 사용한 형태도 의미 전달에 문제는 없을 듯하다.
- 73 역자주\_IDC(International Data Corporation, <http://www.idc.com>)는 IT 업계 시장조사 전문기관이다.
- 74 역자주\_TTM은 제품을 시장에 출시하는 데까지 걸린 총 시간을 말한다. TTM의 의미는 'Time To Market'이지만 원문처럼 Money를 써도 의미 전달에 문제는 없을 듯하다.
- 75 역자주\_이 글에서 ‘반도체 추상화 계층’이라고 쓰인 표현은 모두 하드웨어 추상화 계층 HAL(Hardware Abstraction Layer)을 말한다.

- 76 역자주\_결과론적인 이야기지만 『Rebel Code, Glyn Moody, Basic Books, 2002』 14장에 언급된 마이크 티만의 회상에 따르면 시그너스는 두 번의 큰 성공 기회를 놓친 일이 있다. 첫 번째는 존 길모어가 제안한 x86용 자유 커널을 개발하려던 계획을 포기한 것이고(『역자주 38』 참고), 두 번째는 1995년에 레드햇을 인수하려던 계획을 그만둔 것이다. 그 후 레드햇은 반대로 시그너스를 인수하게 된다(『역자주 83』의 마지막 부분 참고).
- 77 역자주\_런타임(run-time)은 소프트웨어 분야에서 몇 가지 다른 의미로 사용되는데, 여기서 말하는 '런타임 로열티'는 제품 한 개당 지불해야 하는 요금을 말한다. 예를 들어 MS의 RTOS이자 임베디드 시스템 개발 환경인 Windows Embedded는 그 자체가 유료(약 100 달러)인 것은 물론 이를 이용해 개발한 소프트웨어를 RTOS와 함께 임베디드 제품에 탑재해 판매할 때 제품 한 개당 3~15달러 정도의 비용을 MS에 지불해야 한다. 따라서 런타임 로열티는 저가나 보급형 제품의 경우 상당한 수의 잠식 요인이 된다.
- 78 역자주\_132쪽 표의 수익성 및 CAGR 지표 참고.
- 79 역자주\_1997년 1월에 두 회사의 공동 투자를 받았다. 『역자주 04』의 세너제이 비즈니스 저널이 선정하는 최고 성장 비공개회사 100위 안에 시그너스 솔루션즈가 포함된 때가 바로 이 시기다. 오거스트 캐피털의 벤처투자가 앤디 라파포트(Andy Rappaport)는 투자 결정과 관련한 언급에서 오픈 소스가 경제적 불연속성(economic discontinuity)을 창출할 수 있다고 밀하는데, 이러한 불연속성에서 사용자는 사유 소프트웨어에 값을 지불하는 대신 자유 소프트웨어에 대한 서비스나 애드온(add-on)에 돈을 지불한다고 한다(『리눅스 혁명과 레드햇, 로버트 영, 김영사, 2000년』, 209페이지에서 인용).
- 80 역자주\_닥터 수스(Dr. Seuss, Theodor Seuss Geisel, 1904~1991). 이 문구는 미국에서 출판된 유아용 읽기 교재 『Green Eggs and Ham, Dr. Seuss, Random House, 1960』에서 인용한 것이다. 이 책은 주인공 샘(Sam)이 가상의 음식인 초록색 계란 프라이와 햄을 상대방에게 먹어보라고 권하지만 초록색 계란과 햄은 싫다고 거부하는 단순한 대화를 최소한의 단어로 운율을 살려 반복적으로 구성한 것이다. 책의 마지막 부분은 겉모습만 보고 계속 싫다고 우기던 초록색 계란과 햄을 먹어본 후 좋아하게 되어 이전의 부정문들이 모두 긍정문으로 바뀌며 다시 한 번씩 반복되는 구조로 되어있다. 따라서 여기서 인용한 문장은 '시도해보기도 전에 겉모습만 보고 판단하는 것은 좋지 않다'는 의미를 담고 있다. 공교롭게도 이야기 속의 샘은 빨간 우묵 모자(Red Hat)를 쓰고 등장한다.
- 81 역자주\_eCos는 GNUPro처럼 기존의 개발 도구를 모아 발전적으로 구성한 것이 아니라 임베디드 시스템 개발을 위해 처음부터 완전히 새롭게 개발한 RTOS다. eCos는 리눅스 코드와 관계없이 시그너스가 독자적으로 개발한 운영체제다.

- 82 역자주\_여기서 언급한 오픈 소스 아키텍처는 고수준 임베디드 개발 시스템 아키텍처를 말하지만, 단어의 의미 그대로 오토데스트의 오토캐드 아키텍처(Autodesk AutoCAD Architecture) 소프트웨어를 떠올려 보면 구체적인 이해에 도움이 된다. 건축 설계 디자인 소프트웨어 오토캐드 아키텍처처럼 eCos도 프로그램 메뉴에서 모듈을 선택해 결합하는 방식으로 RTOS와 임베디드 애플리케이션을 만들 수 있다.
- 83 역자주\_이 글이 쓰여진 이후부터 현재까지의 eCos 및 관련 시장 상황에 대해 간략히 설명하면 다음과 같다. eCos는 1998년 11월 2일에 첫 번째 제품이 출시되었으며 레드햇이 시그너스 솔루션즈를 인수·합병한 이후의 이름은 Cygnus를 Configurable로 대체한 'Embedded Configurable Operating System'이다. 이 시기의 자료는 <http://web.archive.org/web/19981202212829/http://sourceware.cygnus.com/>에서 참고할 수 있다. eCos는 처음부터 리눅스와 MS 윈도우용이 함께 출시되었는데 첫 번째 제품에서는 윈도우 환경에서만 GUI 설정 툴이 제공되었다. 첫 번째 제품의 매뉴얼과 GUI 설정 화면은 각각 [http://ecos.sourceforge.org/docs-1.1/\(매뉴얼\)](http://ecos.sourceforge.org/docs-1.1/(매뉴얼)), <http://ecos.sourceforge.org/docs-1.1/guides/user-guides/the-four-panes.html#CONFIGURATION-PANE>(설정 도구 화면)에서 참고 할 수 있다. 한편 레드햇은 오픈 소스 임베디드 사업이 수익성이 없다는 이유로 기업용 리눅스 서버 사업에 집중하기 위해 2002년 eCos 사업에서 철수하는데 이때 해고된 개발자들이 2002년 4월 eCos에 대한 상업 지원을 판매하는 스픈오프 회사 eCosCentric을 창업해 개발을 계속한다. eCosCentric은 과거 시그너스와 같은 방식으로 eCosPro 개발자 키트(eCosPro Developer's Kit)와 지원 서비스를 판매하는 사업을 하며, 공개 개발 및 제품은 <http://ecos.sourceforge.org/>에서 제공한다. eCosCentric의 홈페이지는 <http://www.ecoscentric.com/>이다. 사업에서 철수한 레드햇은 2004년 1월 13일 eCos 개발 공동체의 요구를 수용해 eCos 관련 저작권을 모두 FSF로 양도하기로 발표하고 2008년 5월 모든 절차가 완료되었다. 따라서 현재의 eCos는 FSF가 저작권을 갖고 GPL 호환 이용허락(<http://www.gnu.org/licenses/ecos-license.html>)으로 배포되는 자유 소프트웨어다. 오픈 소스 임베디드 사업과 관련해서 레드햇 한국법인은 2004년 10월 12일 한국의 리눅스 배포판 및 임베디드 업체 미지 리서치(<http://web.archive.org/web/19991013040903/http://mizi.co.kr/>)와 전략적 제휴를 통해 임베디드 사업을 추진하기도 했는데, 미지 리서치는 임베디드 시장 선두업체 윈드리버(Wind River)에 2008년 10월 15일 1천6백만 달러(2008년 화폐가치 약 176억 원, 2013년 화폐가치 약 200억 원)의 금액으로 인수·합병되었다. 기술산업 분야 미디어 업체 UBM Tech의 2013년 임베디드 시장 조사 자료([http://images.content.ubmtechelectronics.com/Web/UBMTechElectronics/%7Ba7a91f0e-87c0-4a6d-b861-d4147707f831%7D\\_2013EmbeddedMarketStudyb.pdf](http://images.content.ubmtechelectronics.com/Web/UBMTechElectronics/%7Ba7a91f0e-87c0-4a6d-b861-d4147707f831%7D_2013EmbeddedMarketStudyb.pdf))에 따르면, 단일 제품으로는 안드로이드(Android)와 우분투(Ubuntu), FreeRTOS 등의 오픈 소스 RTOS가 임베디드 시장에서 선두를 지키고 있으며 eCos는 전체 시장의 3% 미만을 점유하고 있는 것으로 보인다. eCos의 최근 소식으로는 2012년 1월 25일 정상 궤도에 오른 러시아 연방 국가의 대기 관측 위성 Chibis-M(치비스-М)이 eCos를 운영

- 체제(<http://chibis.cosmos.ru/trajectory/>, <http://www.thespacereview.com/article/2213/1>)로 사용한 것인데, 이는 〈역자주 21〉과 관련된 재미있는 후일담이 된다. 이 책의 다른 글 ‘첨단의 리눅스’ 마지막 절인 ‘리눅스의 미래’ 부분에 나오는 리누스 토르발스의 짧은 언급도 eCos의 당시 성공 요인이라고 할 수 있다. 시그너스 솔루션즈의 제품 중 eCos와 관련해 추가로 언급해야 할 것으로 Cygwin이 있다. Cygwin(<http://sourceware.org/cygwin/>)은 MS 윈도우 환경에서 리눅스 프로그램을 사용할 수 있게 해주는 애플레이션 프로그램으로 임베디드 시스템 구현을 지원하기 위해 개발되었으며 현재에도 윈도우 시스템에서 리눅스와 유닉스 프로그램을 실행하는 용도로 많이 이용된다. Cygwin은 시그너스가 개발한 소프트웨어 중에서 지금까지 가장 많은 다운로드 수를 갖고 있다. 마지막으로 레드햇의 시그너스 인수·합병과 관련해서 레드햇이 자신과 비슷한 규모의 회사를 인수한 것은 당시의 IT 거품 경제(Dot-com Bubble)가 반영된 것이기도 하다. 합병 당시의 시그너스의 사업규모는 종업원 수 181명, 연 매출 2천만 달러(1999년 화폐가치 약 238억 원, 2013년 화폐가치 약 357억 원)였다. 14달러(1999년 화폐가치 약 1만 6천 원, 2013년 화폐가치 약 2만 4천 원)으로 발행된 레드햇의 주식은 나스닥(NASDAQ) 첫 거래일인 1999년 8월 11일에 시가 23.00달러(1999년 화폐가치 약 2만 7천 원, 2013년 화폐가치 약 4만 1천 원)를 기록한 이래 시그너스와의 인수·합병 이후 최고가를 경신하며 2:1 액면 분할이 이루어진 2000년 1월 10일 이전까지 최고 267.38달러(2000년 화폐가치 약 30만 2천 원, 2013년 화폐가치 약 44만 4천 원)까지 1,100% 이상 상승했다. 시그너스에 투자한 벤처 투자회사들이 합병을 승인한 이유는 투자금 회수를 위한 출구 전략의 하나였다. 2013년 8월 12일 뉴욕증권거래소(NYSE)의 레드햇 종가는 52.28달러(약 5만 6천8백 원)이며 레드햇 주식의 상장 때부터 지금까지의 총 수익률은 101.42%다. 『리눅스 그냥 재미로, 리눅스 토발즈 외, 한겨레신문사, 2001년』의 ‘세상에서 가장 운 좋은 사나이’ 부분에서 레드햇 주가에 대한 짧은 애피소드를 볼 수 있으며, IT 거품 경제에 대해서는 다큐멘터리 영화 「Startup.com, Chris Hegedus, et al., 2001」(<http://www.youtube.com/watch?v=ibuiUXOTE4M>)를 참고할 수 있다.
- 84 역자주\_제임스 조이스(James Joyce, 1882~1941). 이 문구는 1916년에 출판된 소설 『젊은 예술가의 초상(A Portrait of the Artist as a Young Man)』의 도입 부분에서 인용한 것으로 실제로는 로마의 시인 오비디우스(Publius Ovidius Naso)의 서사시 『변신 이야기(Metamorphoses)』 8권에 나오는 것이다. 한국어로 번역된 조이스의 책에서는 arts를 예외 없이 예술로 번역하지만, 『변신 이야기』에서 이 문장이 나온 부분은 다이달로스(Daedalos)가 아들 이카로스(Icaros)와 함께 미궁을 빠져나가기 위해 밀랍으로 날개를 만드는 장면에 해당하기 때문에 오픈 소스 소프트웨어를 다룬 이 글에서는 arts를 ‘기술’ 또는 ‘새로운 발명’으로 옮기는 것이 나을 것 같다. 이 책은 <http://archive.org/details/portraitofartist00joycrich>에서 참고할 수 있다.

# 6 | 소프트웨어 공학

폴 빅시 Paul Vixie

송창훈 역

소프트웨어 공학<sup>01</sup>은 프로그램을 만드는 것보다 광범위한 분야다. 그런데 역사적인 예를 살펴보면 소프트웨어 공학을 적용해야만 인기 있고 널리 사용되는 소프트웨어가 되는 것은 아닌 게 명백하며, 많은 오픈 소스 프로젝트에서 프로그램은 소프트웨어 공학 없이 단순히 만들어져 배포된다. 나는 이 글에서 소프트웨어 공학의 일반적인 요소와 이에 대응하는 오픈 소스 공동체의 일반적인 대체물을 살펴보고 두 가지 접근 방식의 차이와 그러한 차이가 갖는 의미를 설명하려고 한다.

## 소프트웨어 공학 공정

소프트웨어 공학 공정은 일반적으로 다음과 같은 요소로 구성된다.<sup>02</sup>

- |              |          |
|--------------|----------|
| ① 시장 요구사항 분석 | ② 시스템 설계 |
| ③ 상세 설계      | ④ 구현     |
| ⑤ 통합         | ⑥ 현장 테스트 |
| ⑦ 사후 지원      |          |

위의 요소들은 순차적으로 진행되며 한 단계가 완료되어야만 다음 단계로 넘어갈 수 있다. 하나의 요소가 수정된 경우에는 이와 관련된 모든 요소가 재검토되거나 수정되어야 한다. 경우에 따라서는 연관된 요소들이 완전히 기술되기 전에 주어진 요소를 기술하거나 구현하는 것이 가능한데, 이러한 형태를 선행 연구 또는 선행 개발이라고 한다.

소프트웨어 공학 공정의 모든 요소에는 동료검토나 멘토 및 관리 검토, 제휴검토 등의 여러 가지 검토가 필수적으로 이루어져야 한다.<sup>03</sup>

소프트웨어 공학 공정의 요소들은 문서 자료나 소스 코드를 불문하고 모두 판 번호 version 표시와 감사 내역을 포함해야 한다. 어떤 요소에 대한 수정 여부를 결정하기 위해서는 몇 가지 형식의 검토가 필요하며, 검토 수준은 수정이 이루어질 범위와 직접 연관해서 고려해야 한다.

## 시장 요구사항 분석

소프트웨어 공학 공정의 첫 번째 단계는 제품을 사용할 최종 고객과 그들이 제품을 필요로 하는 이유를 자세하게 기술한 문서를 만드는 것이다. 그리고 고객이 원하는 제품의 기능을 차례대로 기술해 간다. 시장 요구사항 분석서인 MRD<sup>Marketing Requirements Document</sup>는 ‘무엇을 개발할 것이며, 그것을 누가 사용할 것인가?’를 결정하는 문서다.

실패한 많은 프로젝트에서는 마케팅 부서에서 만든 MRD가 마치 글을 새긴 비석처럼 공학 부서로 그대로 전달되기 때문에 크립토나이트<sup>04</sup> 같은 이미 존재하는 해결 방법을 갖고 있지 않은 공학 실무자로서는 물리 법칙이 어떻다느니 MRD의 제품은 실제로 만들 방법이 없다느니 하는 불평불만을 끊임없이 쏟아낼 수밖에 없다. MRD는 공학적인 검토와 많은 양의 문서를 작성하는 작업이 마케팅과 함께 결합된 노력을 통해 만들어져야 한다.

## 시스템 설계

시스템 설계 System-Level Design는 제품을 ‘모듈(때로는 ‘프로그램’)'과 모듈 간의 상호 관계의 측면에서 고수준으로 기술하는 것이다. 이 단계에서 만들어진 문서인 SDD<sup>System-Level Design Document</sup>는 첫째, 정상적으로 작동하는 제품을 만들 수 있는 더 나은 신뢰성을 가져야 하고 둘째, 제품을 만드는 데 소요될 작업의 총량을 추정

할 근거가 되어야 한다. 시스템 설계 문서는 고객의 필요와 제안된 시스템 설계를 통해 그러한 필요가 충족되는지 평가할 수 있는 시스템 테스트 계획을 개괄적으로 포함해야 한다.

## 상세 설계

상세 설계 Detailed Design는 시스템 설계 문서에 기술된 (소프트웨어 실행을 위한 명령형 형식과 API 호출 방법, 외부에서 참조할 수 있는 자료구조 등과 같은) 모든 모듈을 구체적으로 기술하는 것이다. 모듈 사이의 인터페이스와 상호 의존성이 모두 이 단계에서 결정된다. 또한 상세 설계를 통해 퍼트 PERT 차트나 갠트 Gantt 차트<sup>05</sup>가 만들어져서 어떤 일이 어떤 순서로 진행되어야 하는 지와 각각의 모듈을 완성하는데 필요한 시간을 보다 정확하게 추정할 수 있어야 한다.

모든 모듈에는 모듈을 구현할 사람에게 해당 기능을 검증해 볼 수 있는 테스트 항목과 테스트 유형을 알려주는 단위 테스트 계획이 포함되어야 한다. 비기능적인 그 밖의 단위 테스트에 대해서는 뒷부분의 ‘테스트 세부 사항’에서 설명한다.

## 구현

상세 설계 문서에 기술된 모든 모듈은 빠짐없이 구현 Implementation되어야만 한다. 구현 단계에서는 소프트웨어 공학 공정의 일부면서 또한 가장 핵심이라고 할 수 있는 프로그래밍 또는 코딩 작업이 이루어진다. 유감스럽게도 코딩은 소프트웨어 공학에 있어 사실상 독학할 수 있는 유일한 부분이기 때문에 때때로 소프트웨어 공학에서 가르치거나 학습되는 유일한 부분이 되어 버린다.

하나의 모듈이 만들어진 뒤에는 다른 모듈이나 시스템 테스트 공정에 의해 성공적으로 검증되어 사용된 이후에야 구현된 것으로 간주할 수 있다. 모듈을 만드는 과정은 소스 코드를 편집하고 컴파일하는 작업을 반복하는 전통적인 방법을 따른다. 모듈 테스트에는 상세 설계에서 기술한 단위 기능 테스트와 회귀 테스트가 포함되며, 성능/스트레스 테스트와 코드 커버리지 분석도 포함된다.<sup>06</sup>

## 통합

모든 모듈이 명목상으로 완성되었다면 시스템 차원의 통합(Integration)이 이루어질 수 있다. 이 과정에서 모든 모듈을 한 곳의 소스 풀(source pool)로 옮긴 뒤에 컴파일과 링크 과정을 거쳐 하나의 시스템으로 만들게 된다. 통합은 다양한 모듈의 구현과 병행해 점진적으로 이루어질 수 있지만, 모든 모듈이 실질적으로 완성되어야만 통합 공정의 종결을 확정할 수 있다.

통합 공정에는 시스템 테스트를 개발하는 것도 포함된다. 빌드된 패키지가 (tarball<sup>107</sup>을 풀거나 CD-ROM에서 파일을 복사하는 것처럼) 설치되어야 하는 형태일 경우는 전용 시스템이나 시뮬레이션 환경에서 그것을 테스트해 볼 수 있는 자동화된 방법이 필요하다.

때때로 미들웨어 분야에서는 패키지가 단지 빌드된 상태의 소스 풀일 수 있는데, 이러한 경우에는 설치 도구가 존재하지 않기 때문에 소스 풀에 대해 시스템 테스트가 이루어진다.

설치 가능한 형태였을 경우, 일단 패키지가 설치된 후에는 자동화된 시스템 테스트에 의해 조합할 수 있는 모든 인수와 함께 public으로 선언된 모든 명령어를 실행할 수 있어야 하고, public으로 선언된 모든 진입점(entry point)을 호출할 수 있어야 한다. 만약 시스템이 데이터베이스 등을 생성하는 것이라면 자동화된 시스템 테스트는 테스트용 데이터베이스를 생성한 후에 별도로 작성된 외부 접근 도구를 이용해 데이터베이스의 무결성을 검증해 볼 수 있어야 한다. 이러한 과정에 단위 테스트가 활용될 수 있게 하는 것도 가능하며 통합과 빌드, 패키징 공정 중에 모든 단위 테스트가 순차적으로 수행되도록 할 수도 있다.

## **현장 테스트**

현장 테스트 Field Test는 일반적으로 내부에서 시작된다. 이 말은 소프트웨어 패키지를 만든 조직의 직원이 그들의 컴퓨터에서 소프트웨어를 먼저 테스트해본다는 의미며, 여기에는 데스크톱 컴퓨터와 노트북, 서버와 같은 출시 가능한 production level 모든 종류의 시스템에 대한 테스트가 포함되어야 한다. 고객에게 새로운 소프트웨어나 기존 소프트웨어의 새로운 판을 사용해볼 것을 권할 때 ‘우리가 직접 사용해 보았다’고 말하고 싶을 것이다. 내부 현장 테스트 과정 동안에는 소프트웨어 개발자의 기술 지원이 직접 제공될 수 있어야 한다.<sup>08</sup>

궁극적으로 고객이나 예상 고객의 컴퓨터에서 소프트웨어를 실행해 보는 외부 현장 테스트가 필요하다. 외부 테스터들은 내부 테스터와 비교할 때 소프트웨어 사용에 대한 다른 습관과 방법을 갖고 있기 때문에 사소하고 명백한 것을 포함한 많은 결함을 발견할 수 있다. 따라서 외부 현장 테스트에는 피드백을 가능하면 많이 제공해줄 수 있는 우호적인 고객을 참여시키는 것이 바람직하다. 소프트웨어 개발자는 외부 현장 테스트 기간 중에도 단계적인 방법을 통해 외부 테스터와 연결될 수 있어야 한다.<sup>09</sup>

현장 테스트 과정에서 발견한 결함은 선임 개발자와 기술 마케팅 담당자에 의해 분류된 후에 문서상으로 수정할 수 있는 것은 무엇이며, 현재의 판이 출시되기 전에 수정해야 할 것과 다음 판에서 수정해야 할 것은 무엇인지, 또한 수정하지 않아도 될 것은 무엇인지 등의 사항이 결정되어야 한다.

## **사후 지원**

현장 테스트나 소프트웨어가 배포된 이후에 발견된 결함은 모두 추적 시스템에 등록되어야 한다. 등록된 결함은 시스템 정의와 문서 또는 모듈의 정의나 구현을 수정할 기술자에게 최종적으로 맡겨져야 한다. 결함을 수정한 후에는 최초의 결함과 그것이 해결되었음을 확인할 수 있는 (그래서 이후의 재발을 방지할 수 있는) 단위

테스트와 시스템 테스트를 함께 또는 선택적으로 회귀 테스트의 형태로 실행해야 한다.

시장 요구사항 분석서인 MRD가 공학과 마케팅의 공동 작업이었던 것처럼 사후 지원 Support 또한 공학과 고객 서비스 사이의 공동 작업이라고 할 수 있다. 이러한 협업의 성공 여부는 발생한 문제를 체계적으로 분류하고 출시 가능한 소프트웨어에 있는 치명적인 결함을 최대한 줄이는 것 등의 버그 관리에 달려 있다.

## 테스트 세부 사항

### 코드 커버리지 분석

코드 커버리지 분석 code coverage analysis은 전처리기 preprocessor나 오브젝트 코드 수정기 object code modifier를 이용하거나 컴파일러나 링커의 특별한 모드를 이용해 프로그램 코드를 계측하는 것으로 시작한다. 소스 코드의 한 블록 안에서 조합할 수 있는 코드의 실행 순서를 코드 경로 code path라고 하는데, 코드 커버리지 분석은 프로그램이 실행되는 동안 어떤 코드 경로가 선택되어 실행되었는지를 확인하고 기록할 수 있어야 한다.<sup>10</sup>

다음과 같은 전형적인 C 코드를 살펴보자.<sup>11</sup>

---

```
① if (read(s, buf, sizeof buf) == -1)
② error++;
③ else
④ error = 0;
```

---

일반적인 테스트 상황에서 ①행의 read() 함수 실행에 오류가 발생하여 -1이 반환된 후에 ②행이 실행돼 error 변수 값이 증가할 가능성은 드물다고 할 수 있다. 그런데 error 변수가 초기화되지 않은 상태에서 ②행이 실행되는 경우가 있다면 어

떤 값인지 모르는 변수의 값이 증가하는 결과가 되므로 이후의 프로그램은 정의되지 않은 예측 불가능한 결과를 가져오는 버그를 포함하게 된다. 이러한 종류의 버그로 인한 사후 지원 비용 발생을 미연에 방지할 방법은 단위 테스트를 통해 가능한 모든 코드 경로를 살펴 올바른 결과가 나오는지 확인하는 것이다.

그러나 코드 경로는 조합되는 복잡한 것이기 때문에 위 예에서 error 변수를 조건 블록 이전에 초기화시키면 에러가 발생하지 않는 좀 더 깔끔한 코드를 만들 수 있다. 다음 코드에는 어떠한 평가 방법으로도 찾을 수 있는 명백한 오류<sup>12</sup>가 포함되어 있다. 코드를 통해 간단한 것이 복잡해지는 게 얼마나 쉬운지 알 수 있다.

---

```
① if (connect(s, &sa, &sa_len) == -1)
②     error++;
③ else
④     error = 0;
⑤ if (read(s, buf, sizeof buf) == -1)
⑥     error++;
⑦ else
⑧     error = 0;
```

---

위 코드로부터 테스트해 볼 수 있는 네 가지 코드 경로를 만들 수 있다.

1. ① – ② – ⑤ – ⑥
2. ① – ② – ⑤ – ⑧
3. ① – ④ – ⑤ – ⑥
4. ① – ④ – ⑤ – ⑧

조합할 수 있는 모든 코드 경로를 테스트해본다는 것은 일반적으로 불가능하다. 왜냐하면 몇 십 행의 코드로 구성된 작은 함수의 경우에도 가능한 코드 경로는 수백 가지 이상이 될 수 있기 때문이다. 또한 단위 테스트가 프로그램이나 모듈의 코드

전체를 연속적인 실행 등으로 모두 확인할 수 있다고 확신하는 것만으로는 충분하지 않다. 이러한 종류의 커버리지 분석 도구는 현장 소프트웨어 기술자 모두가 이용할 수 있는 것이 아니며, 이것이 바로 QA가 나름의 고유 영역을 가질 수 있는 이유가 된다.<sup>13</sup>

### 회귀 테스트

버그를 수정하는 것만으로는 충분하지 않다. ‘관찰로 버그를 찾아낼 수 있다’는 말은 종종 오류를 확실하게 잡아낼 수 있는 테스트 도구를 만드는 것이 어렵기 때문에 변명처럼 사용되기도 한다. 물론 숫자를 0으로 나누는 경우처럼 코드를 관찰하는 것으로도 쉽게 찾을 수 있는 명확한 버그도 많이 존재한다. 그러나 무엇을 수정해야 할지 판단하려면 프로그래머의 의도를 정확히 알기 위해 주변의 코드를 함께 살펴보아야만 한다. 이러한 종류의 분석은 수정이 이루어질 때 문서화하거나 소스 코드에 주석의 형태로 포함시키는 방법 중 한 가지 또는 두 가지 모두로 이루어져야 한다.

그러나 관찰을 통해 버그를 찾을 수 없는 경우가 좀 더 일반적이며 버그 수정이 문제의 원인이 된 곳이 아닌 다른 코드 부분에 이루어지기도 한다. 이러한 경우에는 잘못된 코드 경로나 잘못된 프로그램 상태 등을 검사할 수 있는 새로운 단위 테스트를 만들어 수정한 부분을 검사해 보아야 한다. 테스트와 검토가 끝난 뒤에는 또 다른 수정에 의해 같은 버그가 나중에 다시 나타날 때 고객보다 QA가 먼저 버그를 잡을 수 있도록 새로 만든 단위 테스트 자체에 대한 확인과 검토가 이루어져야 한다.

### 오픈 소스 소프트웨어 공학

오픈 소스 프로젝트는 위 소프트웨어 공학 공정 하나하나를 모두 포함할 수 있는데, 공정하게 말하면 실제로 그런 예가 있다. BSD와 BIND, Sendmail의 상용 제품은 모두 표준적인 소프트웨어 공학 공정이 적용된 사례라고 할 수 있다.

그러나 이 제품들이 처음부터 소프트웨어 공학에 따라 개발된 것은 아니었다. 본격적인 소프트웨어 공학 공정은 흔히 매출 계획 등의 기대 수익이 있어야 투자가 이루어질 수 있는, 자원이 많이 들어가는 실제적인 작업이다.

오픈 소스 프로젝트에서 좀 더 일반적인 경우는 단순히 작업에 재미를 느껴 자신이 만든 프로그램이 널리 사용되기를 바라는 마음으로 무료로 배포하거나 때로는 배포 상의 제한을 두지 않는 것이다. 이런 부류의 사람들은 코드 커버리지 분석이나 범위 검사 인터프리터, 메모리 무결성 검증 도구 등과 같은 소위 ‘상용 등급’의 개발 도구를 갖지 못한 경우가 대부분이다.<sup>14</sup> 또한 이들이 주로 흥미를 느끼는 것은 코딩과 패키징 그리고 오픈 소스의 전파에 있지 QA나 MRD가 아니며 급박한 제품 출하 시기에 얹매여 있지도 않다.

다시 한번 소프트웨어 공학 공정을 살펴보면서 애정과 노력 외에는 개발 자금이 제공되지 않는 오픈 소스 프로젝트에서 어떤 것들이 그 역할을 대신하는지 알아보도록 하자.

## 시장 요구사항 분석

오픈 소스 사람들은 자기가 필요하거나 갖고 싶은 도구를 직접 만드는 경향이 있다. 때때로 이러한 도구는 직업상 필요 때문에 만들어지곤 하는데 소프트웨어 공학 보다는 시스템 관리와 같은 일에 종사하는 사람에 의해 만들어지는 경우가 많다. 몇 번의 과정이 반복되면 소프트웨어의 크기가 상당히 커지면서 스스로 생명력을 갖게 되어 인터넷을 통해 tarball로 배포되기에 이른다. 또한 다른 사용자가 소프트웨어에 추가되길 원하는 기능을 요청하거나 여러 기능을 직접 구현해 보내오기도 한다.

오픈 소스 MRD가 만들어지는 곳은 대부분 메일링리스트나 뉴스그룹이며 사용자와 개발자가 함께 직접 의견을 교환한다. 합의는 개발자가 동의하거나 기억하는 것

으로 이루어지며, 합의에 이르지 못하면 자신이 원하는 독립된 판을 만들기 위해 코드가 분리되는 결과로 이어진다.<sup>15</sup> 오픈 소스에서도 MRD에 해당하는 것을 잘 만들 수 있지만 첨예한 부분도 있다. 의견 대립 때문에 합의점을 찾을 수 없는 경우가 종종 있고 합의를 도출하려는 시도가 이루어지지 않을 때도 있다.

## 시스템 설계

일반적으로 개발 자금 투자가 없는 오픈 소스 개발에는 시스템 수준의 설계가 없다고 보는 것이 타당할 듯싶다. 오픈 소스에서의 시스템 설계는 함축된 형태로 존재하다가 어느 날 갑자기 완성된 형태가 나타나거나, 아니면 소프트웨어 본체와 함께 점진적으로 발전되는 형태가 된다. 오픈 소스 시스템이 2판 또는 3판에 이르면 문서화가 전혀 이루어지지 않았다 하더라도 대부분 실제적인 시스템 디자인을 갖추게 된다.

바로 이러한 점이 오픈 소스가 소프트웨어 공학의 일반적인 기준으로부터 출발한 다른 소프트웨어에 비해 신뢰성이 떨어진다는 평가를 받는 원인이 된다. 정식 MRD나 심지어 정식 QA의 부족이 뛰어난 프로그래머나 정말로 호의적인 사용자를 통해 보상될 수 있더라도, (누군가의 머릿속에는 들어 있다고 해도) 시스템 디자인이 없는 프로젝트의 품질은 자체적인 한계를 갖게 된다.

## 상세 설계

개발 자금 없이 재미를 추구하는 특성 때문에 생기는 또 한 가지 희생물이 상세 설계다. DDD Detailed Design Document에 흥미를 갖는 사람도 있지만, 이러한 사람은 직업과 관련해서 근무 시간 중에 DDD를 작성하려고 할 뿐이다. 또한 상세 설계는 독립 공정이 아닌 구현의 부산물이 되어 버리기도 한다. “이 작업에는 파서가 필요하니 나중에 하나 만들어야겠어”라고 판단하는 경우처럼 API의 외부 기호 external symbol 정보를 헤더 파일이나 맨 페이지<sup>16</sup> 안에 문서화하는 작업은 선택적인 것이며,

API를 출판하거나 이를 프로젝트 외부에서도 이용할 수 있게 하려는 의도가 없다면 상세 설계 단계에서 이러한 종류의 문서 작업은 이루어지지 않을 수 있다.

이것은 부끄러운 일이다. 왜냐하면 재사용될 수 있는 많은 유용한 코드가 이런 방식으로 인해 사라져버리기 때문이다. 심지어 재사용되지 않거나 만들어질 때부터 해당 프로젝트에 단단히 결합된 모듈이라서 그 모듈의 API가 미래에 사용되지 않을 것이라 해도 맨 페이지에는 그 기능에 대한 설명과 호출 방법이 기술되어 있어야만 한다. 이는 코드를 향상시키려는 다른 사람에게 매우 큰 도움이 될 수 있다. 왜냐하면 바로 이러한 정보를 읽고 이해하는 것으로부터 작업이 시작될 수 있기 때문이다.

## 구현

구현은 재미있다. 구현은 프로그래머가 가장 사랑하는 것이며 졸음을 쫓아가며 밤 늦도록 해킹에 몰두하게 만드는 것이기도 하다. 코드를 작성할 기회는 거의 모든 오픈 소스 소프트웨어 개발에 있어 참여의 우선적인 동기가 된다. 만약 소프트웨어 공학에서 다른 모든 요소를 배제하고 구현에만 초점을 맞춘다면 여기에는 대단히 많은 표현의 자유가 있다.

오픈 소스 프로젝트는 대부분의 프로그래머가 새로운 스타일을 시험하는 곳이다.<sup>17</sup> 소스 코드 작성에 있어 들여쓰기 형식이나 변수 이름 짓는 방법 또는 ‘메모리를 줄이기 위해’ 또는 ‘CPU 사이클을 줄이기 위해’ 그리고 원하는 어떠한 효율을 높이기 위한 시도도 이루어진다. 아름다움을 가진 멋진 작품들이 tarball로 도처에 깔려 있고, 자신만의 새로운 스타일을 만들어 동작하게 하려는 프로그래머들이 그곳에 있다.

개발 자금 지원이 없는 오픈 소스 활동은 원하는 만큼의 엄격함과 지속성을 가질 수 있다. 대부분의 사람은 개발자가 구현 과정에서 스타일을 세 번 바꾼다 해도 개의치 않는다. 실용적인 코드라면 사용자들은 상관없이 사용할 것이다. 그러나 개

발자는 일반적으로 스타일에 신경을 쓰거나 시간이 지나면 신경을 쓰게 된다. 이런 상황에 있어 래리 월 Larry Wall의 ‘예술적 표현이 되는 프로그래밍’이라는 예전 언급은 매우 적절한 것이다.<sup>18</sup>

개발 자금 지원이 없는 오픈 소스 구현의 주된 차이점은 검토가 비공식적으로 진행 된다는 것이다. 완성되기 전에 멘토나 동료가 코드를 살펴보는 일은 일반적으로 거의 없으며, 단위 테스트나 회귀 테스트와 같은 것도 존재하지 않는다.

## 통합

오픈 소스 프로젝트에서 통합은 일반적으로 맨 페이지를 작성하고, 개발자가 사용 할 수 있는 모든 종류의 컴퓨터에서 올바르게 빌드되는지 확인하고, Makefile을 이용해 구현 과정에서 만들어진 임시 코드 잔존물을 깨끗이 없애고, README 파일을 만들고, 하나의 tarball 파일로 묶어 익명 FTP 사이트에 올린 뒤에 프로그램에 관심을 가질 만한 사람이 알 수 있게 메일링리스트나 뉴스그룹에 소식을 전하는 것이다.

comp.sources.unix 뉴스그룹은 룰 브라운 Rob Brown에 의해 1998년에 다시 활성화됐는데, 새로운 오픈 소스 소프트웨어 패키지나 업데이트 정보는 이곳에 올리는 것이 매우 좋다. 이곳은 저장소나 공공 자료 보관소 archive의 역할을 하기도 한다.<sup>19</sup>

오픈 소스 개발에는 (열이나 PostgreSQL 같은 예외가 있지만) 시스템 테스트 계획이나 시스템 테스트, 단위 테스트가 일반적으로 없다. 오픈 소스 개발의 경우 대체로 테스트에 별로 비중을 두지 않는 편이다. 그러나 다음에 설명할 이유로 인해 제품 출시 전에 테스트를 하지 않는다는 것이 단점이 되지 않는다.

## 현장 테스트

NASA의 우주 로봇 테스트<sup>20</sup>를 비교 대상으로 삼지 않는다면, 오픈 소스 소프트웨어는 산업계에서 가장 좋은 시스템 테스트 환경을 가졌다고 할 수 있다. 그 이유는 사용자는 돈을 지불할 필요가 없을 때 프로그램에 더욱 친근히 다가서는 경향이 있고, (흔히 개발자 자신을 포함하는) 고급 사용자들은 자신이 실행하는 프로그램의 소스 코드를 읽고 고칠 수 있는 경우에 훨씬 많은 도움을 주기 때문이다.

현장 테스트의 본질은 엄격함의 결여다. 소프트웨어 공학이 현장 테스터에게 요구하는 것은 시스템이 설계되고 빌드될 때 태생적으로 예측할 수 없었던 사용 습관에 대한 것이다. 다시 말하면 실제 사용자의 실제 사용 경험이다. 오픈 소스 프로젝트는 이 부분에 있어 그야말로 강점을 가진다.

오픈 소스 프로젝트가 누리는 또 하나의 이점은 단순히 패키지에 포함된 프로그램을 실행하는 것이 아닌 소스 코드를 읽는 방법으로 버그를 찾는 다른 수많은 프로그래머에 의한 동료검토다. 어떤 사람은 보안상의 결함을 찾으려 노력하고 발견된 결함 중 일부는 (다른 크래커<sup>21</sup>들 외에는) 보고되지 않기도 한다. 그러나 이러한 위험이 소스 코드를 읽는 수많은 신참자가 가져다 주는 전체적인 이점을 없애진 못한다. 신참자들은 기존의 어폐한 관리자나 멘토도 하지 못했던 것이 가능하도록 오픈 소스 개발자를 실제로 지원할 수 있다.

## 사후 지원

사용자가 버그를 발견했을 때 개발자에게 들을 수 있는 말은 흔히 “이런, 미안합니 다!” 정도고 버그 패치를 만들어 개발자에게 보낸 경우에도 “아! 미안해요. 고맙습니다!” 정도의 대답을 들을 수 있을 뿐이다.

개발자가 버그 패치에 별로 신경 쓰고 싶지 않은 경우라면 “저는 별문제 없이 잘 되는데요”라고 답하게 된다. 오픈 소스에서 흔히 볼 수 있는 이런 상황이 혼란스럽게

느껴질 수도 있다. 사후 지원의 결여는 오픈 소스 프로그램에 다가서려는 사람의 숫자를 줄이는 원인이 될 수도 있겠지만, 이러한 점이 컨설팅 업체나 소프트웨어 배포업체로 하여금 지원 서비스와 향상된 상용판을 만들어 판매할 기회를 준다.

유닉스 공급업체들이 기본 시스템에 오픈 소스 소프트웨어를 포함해 제공해 달라는 소비자의 강한 요구에 직면했을 때 그들의 처음 반응은 ‘제공은 하겠지만 오픈 소스에 대한 지원은 하지 않겠다’는 것이었다. 시그너스 솔루션즈와 같은 회사의 성공은 이러한 정책을 재검토한 데서 찾을 수 있다. 그러나 오픈 소스와 기업 사이에는 아직도 상당한 문화 차이가 존재한다. 유닉스 공급업체를 포함한 전형적인 소프트웨어 회사들은 무수히 많은 낯선 사람의 기여로 검토되지 않은 수정이 이루어 질 경우에는 지원 사업의 판매 가격 산정을 위한 계획이나 예산을 세울 수 없다.

때때로 이러한 문제에 대한 해답은 단위 테스트와 시스템 테스트, 코드 커버리지 분석 등을 포함한 정상적인 QA 공정을 거쳐 소프트웨어를 내부화하는 것이다. 이러한 작업에서는 QA 공정에서 어떤 기능을 테스트해야 할지 찾아내기 위해 프로그램 코드역분석, 즉 리버스 엔지니어링<sup>reverse engineering</sup>으로 MRD와 DDD를 만들어야 할 수도 있다. 또 다른 해결 방법은 지원 계약서의 ‘확실한 보증을 제공한다’는 조항을 ‘최선의 노력을 다한다’라는 하향된 용어로 바꾸는 것이다. 오픈 소스 소프트웨어 지원 시장은 결국 볼특정 다수로부터 지렛대 효과<sup>leverage</sup>를 끌어낼 수 있는 업체가 차지하게 될 것이다. 왜냐하면 그들 중 많은 사람은 좋은 소프트웨어를 만드는 뛰어난 사람들이고, 오픈 소스 문화는 대부분의 경우 사용자가 실제로 원하는 수준의 기능을 만들어 내는데 매우 효과적이기 때문이다(리눅스와 윈도우의 비교가 이를 증명한다).

## 결론

공학은 오래된 기술 분야며 소프트웨어뿐 아니라 하드웨어를 만들거나 철교를 건설할 때도 근본적으로 다음과 같은 동일한 공학 공정을 거친다.

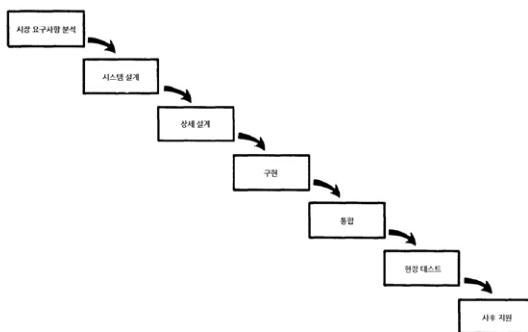
- 요구사항과 그 수요자를 파악한다.
- 요구사항을 충족시킬 해결 방법을 설계한다.
- 설계를 모듈화하고 구현 계획을 세운다.
- 구현과 테스트, 납품, 사후 지원을 제공한다.

몇몇 분야는 일부 공정에 특히 중점을 두기도 한다. 예를 들어 철교 시공업체는 MRD나 구현 공정, 사후 지원에 대해서는 보통 많은 관심을 두지 않지만 SDD와 DDD 그리고 QA에는 매우 면밀한 주의를 기울인다.

공학이 하나의 독립 분야라는 인식과 이 분야에서는 기본적으로 다른 사고방식과 더 많은 작업이 요구된다는 것을 깨달을 때, 프로그래머는 소프트웨어 공학자로 발돋움할 수 있다. 오픈 소스 개발자는 흔히 수년의 성공이 지나간 뒤에야 비로소 프로그래밍과 소프트웨어 공학의 차이점을 알게 되는데, 이것은 오픈 소스 프로젝트가 공학의 엄밀함 없이 오랫동안 지속되었던 관성 때문이다.

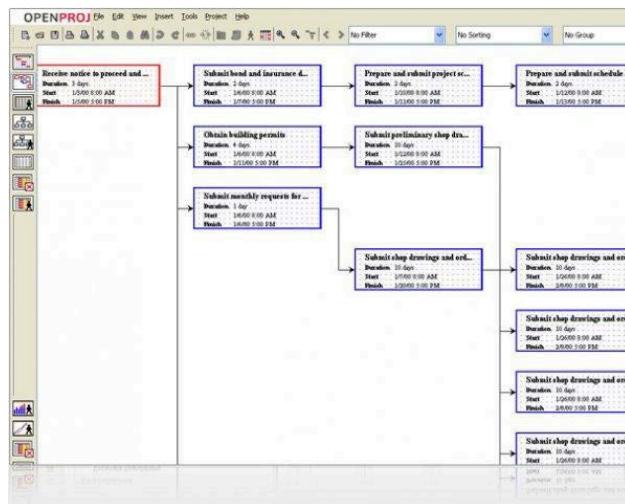
이 글은 소프트웨어 공학의 매우 단편적인 윤곽만을 소개했지만, 이 분야에 입문하려는 오픈 소스 개발자에게 조금의 동기와 지식을 제공했기를 희망한다. 또한 미래는 과거와 현재에 쏟아온 최선의 것들이 함께 결합해 이루어진다는 사실을 기억하기 바란다. 소프트웨어 공학은 단지 계산자와 주머니 보호낭을 위한 것이 아니다.<sup>22</sup> 소프트웨어 공학은 ‘한 명의 특출 난 프로그래머’에 의해 좌우되는 일반적인 오픈 소스 프로젝트의 접근 방식에 의존하지 않고도 고품질의 시스템을 만들 수 있는 검증된 수많은 기술이 있는 풍부한 분야다.

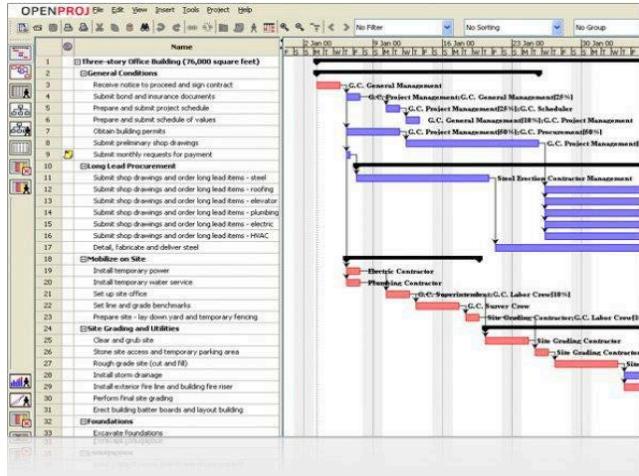
- 
- 01 역자주\_소프트웨어 공학(software engineering)이란 용어는 1968년에 처음 등장했는데(<http://homepages.cs.ncl.ac.uk/brian.randell/NATO/NATOREports/>) 소프트웨어의 개발과 유지관리 등에 공학적인 방법을 도입해 기존의 문제점을 해결하려는 방법론적인 접근 방식을 말한다. 한국에는 1990년대 중반부터 본격적으로 소개되었다. 이 책의 전체적인 흐름과 관련해서 유닉스가 처음 발표된 때가 1969년, C 언어가 발표된 것이 1972년인 것을 고려하면 소프트웨어 공학은 더 체계적인 현대적 소프트웨어의 설계라는 시대적 요청과 맥을 같이 힘을 알 수 있다. 이 책의 다른 글에서 자주 언급되는, 소프트웨어 공학론에 대한 가장 널리 알려진 저작인 『The Mythical Man-Month, Frederick Brooks, Addison-Wesley, Anniversary Edition, 1995』 외에 추천할만한 개론서로 『Software Engineering, Ian Sommerville, Addison-Wesley, 9th Edition, 2010』이 있다. 두 책은 각각 다음의 한국어판으로 번역·출판되었다. 『맨 먼스 미신, 프레더릭 브룩스, 케이앤피 IT, 2007년』, 『소프트웨어 공학, 8판, Ian Sommerville, 교보문고, 2008년』.
- 02 역자주\_대표적인 소프트웨어 공학 공정 모델은 폭포수 모델(waterfall model)이다. 순차적으로 진행되는 이 모델은 1970년에 발표된 윈스턴 로이스(Winston Royce, 1929~1995)의 논문 「Managing the Development of Large Software Systems: Concepts and Techniques」([http://leadinganswers.typepad.com/leading\\_answers/files/original\\_waterfall\\_paper\\_winston\\_royce.pdf](http://leadinganswers.typepad.com/leading_answers/files/original_waterfall_paper_winston_royce.pdf))를 통해 소개된 이후 가장 기본적인 소프트웨어 개발 모델로 사용되고 있다.



- 03 역자주\_동료검토(peer review)는 비슷한 수준과 역할을 가진 사람이 프로그램 소스 코드를 분석하는 등의 방법으로 세부 사항을 평가·검토하는 것이며 멘토 및 관리 검토(mentor/management review)는 일종의 컨설팅처럼 상위 접근 능력을 갖춘 사람에 의한 검토를 말한다. 제휴검토(cross-disciplinary review)는 외부나 다른 분야 전문가의 검토를 받는 것이다.

- 04 역자주\_크립토나이트(Kryptonite)는 만화(영화) 슈퍼맨에서 슈퍼맨의 고향 행성 크립톤(Krypton)이 파괴되면서 생성된 가상의 광물질 이름이다. 크립툰인에게만 반응하는데 여러 색깔 중 초록색은 크립툰인의 모든 능력을 무력화시킨다. 이 글에서는 '미리 준비된 해결 방법'을 뜻하는 은유로 사용되었다. 만화(영화) 슈퍼맨 속에 묘사된 크립토나이트의 화학 성분과 흡사한 광물이 2006년 11월 세르비아에서 발견돼 화제가 되기도 했는데, 이 광물은 발견 장소의 이름을 따서 자다라이트(Jadarite)([http://www.nhm.ac.uk/about-us/news/2007/april/news\\_11392.html](http://www.nhm.ac.uk/about-us/news/2007/april/news_11392.html))라고 명명되었다.
- 05 역자주\_PERT(Program Evaluation and Review Technique) 차트는 프로젝트를 진행하는 데 필요한 단위 작업과 소요 일정 등을 노드와 간선으로 연결한 것으로 네트워크 분석 차트라고도 불린다. 퍼트 차트는 작업 사이의 우선 순위와 의존 관계, 프로젝트에 필요한 최소 일정 등을 산정하는 데 사용할 수 있다. 갠트 차트는 헨리 갠트(Henry Gantt, 1861~1919)가 제안한 막대 모양의 차트로 프로젝트 진행 계획을 시간과 목적의 두 가지 관계로 구성한 것이다. 갠트 차트는 일정 계획, 진행 관리, 실적 차트 등에 이용할 수 있다. 가장 널리 알려진 프로그램인 'MS 프로젝트' 외에 리눅스와 MS 환경에서 모두 사용할 수 있는 오픈 소스 프로젝트 관리 소프트웨어 OpenProj(<http://sourceforge.net/projects/openproj>)를 이용하면 퍼트 차트와 갠트 차트를 생성할 수 있다. 다음 그림은 차례대로 OpenProj가 생성한 퍼트 차트와 갠트 차트의 예다.





- 06 역자주\_스트레스(stress test)는 테스트의 한계를 극대화하는 내구성 시험이며, 회귀 또는 재발 테스트(regression test)는 수정이나 확장이 이루어질 때마다 이미 테스트했던 모듈이나 프로그램을 다시 테스트해 오류 여부를 재확인하는 것이다. 코드 커버리지 분석(code coverage analysis)은 본문 뒷부분의 '테스트 세부 사항'에서 자세히 설명한다. 이 책의 다른 글 '시그너스 솔루션즈의 미래' 중 <[역자주 65](#)>에서 언급한 DejaGNU(<http://www.gnu.org/software/dejagnu/>)를 회귀 테스트 도구로 이용할 수 있으며 그 밖의 오픈 소스 테스트 도구에 대한 정보는 <http://www.opensourcetesting.org/>에서 참고할 수 있다.
- 07 역자주\_tarball(타르볼)이란 용어는 여러 개의 파일을 하나로 묶어주는 유닉스/리눅스 프로그램인 tar(Tape ARchive)를 이용해 만든 묶음 파일을 지칭할 때 흔히 사용된다. 인터넷을 통해 배포되는 파일 중 shar(SHeLL ARchive)을 확장자로 갖는 것이 있는데, 이는 MS 윈도우 환경의 자동 압축 풀림 파일처럼 tarball을 자동으로 풀거나 설치할 수 있는 셸 스크립트다.
- 08 역자주\_내부 현장 테스트를 보통 알파 테스트(alpha test)라고 표현하기도 한다.
- 09 역자주\_외부 현장 테스트를 보통 베타 테스트(beta test)라고 표현하기도 한다.
- 10 역자주\_커버리지란 작성한 코드에서 실제로 실행된 부분이 어느 정도인가를 말한다. 즉 코드 실행 노출 범위를 의미한다. 유닉스와 리눅스 환경에서 할 수 있는 가장 간단한 코드 커버리지 분석은 GCC 컴파일러와 함께 제공되는 gcov(Gnu code COVerage) 명령어를 이용하는 것이다. 예를 들어 다음과 같은 옵션으로 gcc를 실행한 뒤에 gcov를 실행하면 filename.c.gcov라는 이름의 텍스트 파일이 만들어지는데, 이 파일의 내용을 참고하면 실행된 코드 경로와 조건 분기가 일어난 곳 등의 정보를 참고할 수 있다. gcov에 대한 좀 더 자세한 사항은 <http://korea.gnu.org/manual/release/gcov/>를 통해 참고할 수 있다.

---

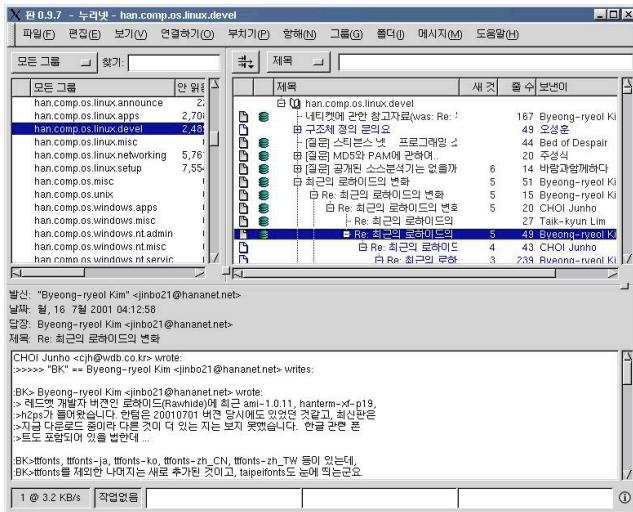
```
natassa@gnu:~$ cat > test.c
#include <unistd.h>
int main(int argc, char **argv) {
    int error, s;
    void * buf;
    if(read(s, buf, sizeof buf) == -1) error++;
    else error=0;
    return 0;
}
Ctrl + D
natassa@gnu:~$ gcc -fprofile-arcs -ftest-coverage test.c
natassa@gnu:~$ ./a.out
natassa@gnu:~$ gcov -b test.c
natassa@gnu:~$ cat test.c.gcov
```

---

11. 역자주\_이 코드는 <[역자주 10](#)>에 있는 실제 실행 코드를 통해 확인해 볼 수 있다.
12. 역자주\_주어진 첫 번째 코드에서 사용된 read()는 파일의 내용을 읽는 함수인데 파일 읽기에 실패하면 -1을 반환한다. 따라서 파일 읽기에 실패할 경우 error 변수에 오류가 발생했다는 의미 (또는 오류의 횟수인) 1을 넣고 그렇지 않을 경우 오류가 없다는 의미로 0을 넣게 된다. 일반적인 조건에서 프로그램이 파일을 읽어 들이는 데 실패하는 경우는 흔치 않기 때문에 error 변수의 값을 초기화하지 않은 코드라 하더라도 테스트 과정에서 오류를 발견하지 못할 수도 있다. 따라서 가능한 오류를 모두 잡아내기 위해서는 조합할 수 있는 모든 코드 경로를 테스트해 보아야 하지만 주어진 코드에서는 error 변수의 값을 초기화하는, 즉 'error = 0;'이라는 한 행을 코드의 시작 부분에 추가하는 것으로 오류 없는 코드를 만들 수 있다. 주어진 두 번째 코드는 같은 상황이 두 번 반복된 경우다. ④행이 실행되면 error 변수가 초기화되는 효과가 있기 때문에 그 뒤의 코드는 어떤 오류도 포함하지 않는다. 그러나 ②행이 실행되었다면 error 변수가 값을 갖게 되기 때문에 그 뒤에 ⑧행이 실행되어 error의 값이 0이 되면 ②행이 기록한 오류도 없어져 버리는 잘못된 코드가 된다. ②행이 실행된 후에 ⑥행이 실행되는 경우는 주어진 첫 번째 코드와 같은 문제를 갖게 된다.
13. 역자주\_QA는 소프트웨어의 결함을 찾기 위한 테스트뿐 아니라 개발 공정 자체를 모니터링하고 개선하는 작업과 정해진 표준을 지켰는가와 테스트 중에 문제가 발생하지 않았는가와 같은 소프트웨어 개발 공정 관리를 포괄하는 분야다. 흔히 테스트와 테스트 공학(Test Engineering, TC), 품질관리(Quality Control, QC), 품질보증(Quality Assurance, QA)을 같은 의미로 쓰기도 하지만 테스트보다는 QC가, QC보다는 QA가 더 넓은 분야라고 할 수 있다. QA에는 지속적인 사후 유지보수 개념이 더 강화되어 있다.

- 14 역자주\_범위 검사 인터프리터(bounds-checking interpreter) 또는 컴파일러는 배열의 범위를 벗어난 접근이 있는지를 확인할 수 있는 인터프리터나 컴파일러를 말한다. <http://boundschecking.sourceforge.net/>에서 GCC에 대한 범위 검사 패치 정보를 참고할 수 있다. 소프트웨어에서의 메모리 무결성 검증(memory integrity verify)은 스택 오버플로(stack overflow)와 같은 허용되지 않은 메모리 접근이 가능한 한지를 확인하는 것이다. <http://www.research.ibm.com/trl/projects/security/ssp/>에서 이와 관련한 GCC의 SSP(Stack Smashing Protector) 기능에 대해 참고할 수 있다.
- 15 역자주\_이러한 상황을 흔히 프로젝트 포킹(project forking) 또는 분기라는 말로 표현한다.
- 16 역자주\_유닉스나 리눅스 시스템에서 man 명령어를 실행했을 때 참고할 수 있는 매뉴얼 문서를 흔히 맨 페이지(manpages)라는 말로 표현한다. <http://man7.org/linux/man-pages/man3/end.3.html>에서 API 기호 정보를 맨 페이지 안에 기술한 예를 참고할 수 있다.
- 17 역자주\_체계적이고 지속적인 소프트웨어의 개발과 유지·관리를 위해서는 통일된 코딩 방법이 필요하기 때문에 다음과 같은 몇 가지 기준 문서가 존재한다. GNU 코딩 표준(<http://www.gnu.org/prep/standards/>), 리눅스 커널 코딩 스타일(<https://www.kernel.org/doc/Documentation/CodingStyle>), 구글 코딩 스타일 가이드(<http://code.google.com/p/google-styleguide/>)
- 18 역자주\_예술적 표현(artistic expression)이란 1997년 8월 20일에 열린 제2회 펄(Perl) 콘퍼런스에서 행해진 래리 월의 기조연설을 통해 대중적으로 소개된 말로 <http://web.archive.org/web/19981203105107/http://www.wall.org/~larry/keynote/keynote.html>의 ‘Artistic Beliefs’ 부분에서 참고할 수 있다. <http://g2.songline.com:8080/ramgen/ntserver1/perl-com/larry-tpc2.rm>에서는 1998년 8월 25일에 열린 제2회 펄 콘퍼런스의 기조연설을 들을 수 있다. 이 연설은 <http://www.wall.org/~larry/onion/onion.html>에서 문서로 참고할 수 있는데, 내용을 다듬어 출판한 것이 바로 이 책에 포함된 ‘근면, 인내, 그리고 겸손’이다. 1999년에 열린 제3회 펄 콘퍼런스에서는 좀 더 발전된 사색의 결과를 엿볼 수 있는 연설이 행해졌는데 <http://www.wall.org/~larry/onion3/talk.html>을 통해 참고할 수 있다. 래리 월은 1998년 제1회 자유 소프트웨어 재단상을 수상하기도 했다. 펄 공동체가 사용하는 이용허락 방식인 ‘예술적 이용허락(Artistic License)’은 작품 안에서 문법이나 고정된 형식에 어긋나는 작가의 예술적 표현을 허용하는 ‘시적 허용’ 또는 ‘예술적 허용’이란 중의적인 의미를 갖고 있다.
- 19 역자주\_1990년대까지 공동 관심사를 함께 논의하는 인기 있는 방법 중 하나가 메일을 주고받듯이 사용하는, 뉴스그룹(newsgroup) 또는 유즈넷(usenet)이라 불리는 텍스트 기반의 통신이었다. 그러나 이 책이 출판된 뒤인 2000년대가 되면서 웹과 웹2.0의 폭발적인 확산으로 인해 뉴스그룹은 이제 고퍼(Gopher), 아치(Archie), PC 통신 등과 마찬가지로 그 자취를 감춰가고 있다. 한국에서는 다른 뉴스그룹들이 침체돼 가

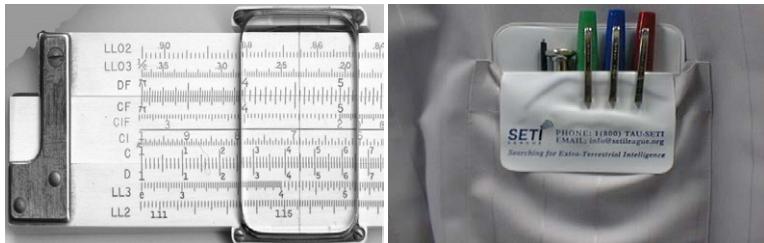
는 도중에도 동영상과 음악 파일 교환을 주목적으로 하는 binary 그룹이 계속 유지되었으나 이 또한 토렌트와 웹 하드가 그 역할을 대신하게 됐다. 뉴스 서버들은 서로 연동되어 필요한 자료를 주고받는 구조로 되어 있기 때문에 한 곳의 뉴스 서버만으로는 정상적인 서비스를 제공하기 어려우며, 현재 한국에는 대중적으로 이용할 수 있는 유즈넷 서비스를 제공하는 곳이 없다. 본문에서 언급한 comp.sources.unix 뉴스그룹의 과거 자료는 <http://ftp.sunet.se/pub/usenet/ftp.uu.net/comp.sources.unix>에서 참고할 수 있으며, 구글 그룹스 서비스(<https://groups.google.com/forum/#forum/comp.sources.unix>)를 통해 예전 글을 참고할 수 있다. 리눅스 개발 그룹의 이름은 comp.os.linux.development이며, 한글 뉴스그룹은 han.comp.os.linux.devel과 같이 han을 최상위 계층으로 가진다. 현재는 뉴스그룹 대신 CMS(Content Management System) 소프트웨어를 기반한 웹 사이트들이 자유 소프트웨어(<http://savannah.gnu.org/>)와 오픈 소스 소프트웨어(<http://sourceforge.net/>)의 개발을 주도하기 때문에 뉴스그룹 보다는 이러한 곳을 이용하는 것이 바람직하다. 다음 그림은 뉴스그룹 사용자 프로그램으로 뉴스그룹을 이용하던 옛날 예요.



20. 역자주\_오래된 자료지만 이 글의 내용과 흐름에 대응되는 미국항공우주국 NASA(National Aeronautics and Space Administration)의 소프트웨어 개발 공정 관련 문서를 <http://homepages.inf.ed.ac.uk/dts/pm/Papers/nasa-approach.pdf>에서 참고할 수 있다. 'Section 10. Keys to Success'는 소프트웨어 개발에 대해 읽어볼 만한 좋은 지침이다.

21. 역자주\_이 책의 다른 글 'GNU 운영체제와 자유 소프트웨어 운동'을 <[역자주 02](#)> 참고

22 역자주\_계산자(slide rule)(<http://www.stefanv.com/calculators/aristo970/>)는 로그의 원리를 응용해 17세기에 발명된 자 모양의 계산 도구로 다양한 종류의 연산을 쉽게 할 수 있다. 계산자는 특히 컴퓨터 대중적으로 보급되기 전까지 공학 분야에서 널리 사용됐는데, 1972년에 본격적인 휴대용 전자계산기 HP-35(<http://www.hpmuseum.org/hp35.htm>)가 출시된 이후부터 급속히 사라져 갔다. 시대적으로 볼 때 엠파이어 스테이트 빌딩(Empire State Building)과 같은 최첨단 건축물의 설계나 최초의 달 착륙 우주선 아폴로 11호의 승무원이 사용한 계산 도구도 모두 전자계산기가 아닌 계산자였다. 주머니 보호낭(pocket protector)은 와이셔츠 주머니에 필기구나 드라이버와 같은 끝이 뾰족한 물건을 넣을 때 주머니가 뚫어지는 것을 막기 위해 주머니에 넣거나 뺄 수 있게 만든 필기구 수납용품이다. 미국이나 유럽에서는 1970년대 까지의 전형적인 공학 실무자의 모습을, 주머니 보호낭을 와이셔츠 주머니에 끼고 계산자를 손에 들고 안전 모를 머리에 쓴 모습으로 묘사하기도 한다. 계산자는 과거의 공학 실무에 있어 만능 도구와도 같은 것이다. 따라서 이 글에서 은유적으로 사용한 '계산자와 주머니 보호낭만을 위한 것은 아니다'라는 말의 의미는 현재의 소프트웨어 공학은 (한 명의 특출 난 프로그래머와 같은) 하나님의 도구 또는 기술에만 의존하는 것이 아닌 더 깊고 다양한 최신 기술이 있는 분야라는 것을 뜻한다(아래 계산자 사진은 자유롭게 이용할 수 있는 공영역 자료(public domain)이며, 주머니 보호낭의 사진 저작권은 외계 지적생명체 탐사 프로젝트인 SETI League에 있다).



## 7 | 첨단의 리눅스

리누스 토르발스 Linus Torvalds

이만용 역

오늘날 리눅스는 수백 만의 사용자와 수천 명의 개발자, 그리고 성장하는 시장을 가지고 있다. 리눅스는 임베디드 시스템에서 사용되기도 하고 로봇 장치를 제어하는 데 사용되기도 한다. 그리고 우주선에 실리기도 했다. 나는 이미 이런 일이 일어날 것이라고 예상하고 있었으며, 이것은 모두 세계 정복 world domination 계획의 일부였다고 말하고 싶다. 하지만 솔직히 말해 이 모든 사실에 약간 놀랐다. 원래 백 명의 사용자에서 백만 명의 사용자로 늘었을 때보다 한 명의 사용자에서 백 명의 사용자로 늘어났을 때 더욱 실감나기 마련이다.

리눅스는 폭넓게 이식하고 사용할 수 있도록 만들고자 했던 원래의 목표 때문이 아니라 훌륭한 설계 원칙과 좋은 개발 모델 기반 때문에 성공할 수 있었다. 이 튼튼한 기반이 이식성과 유용성을 쉽게 달성할 수 있도록 해준 것이다.

잠시 상업적으로 강력한 프로그래밍 언어인 자바나 운영체제인 윈도우 NT와 리눅스를 비교해보자. 자바에 대한 열광은 많은 사람으로 하여금 “한 번 작성하여 모든 곳에서 실행한다” write once, run everywhere .”는 목표가 가치 있는 것이라고 확신하도록 만들었다. 우리는 컴퓨팅 분야에서 더더욱 폭넓은 하드웨어가 사용되는 시대로 이동하고 있으며 따라서 그 목표는 중요한 가치다. 그러나 썬 Sun이 “한 번 작성하여 모든 곳에서 실행한다.”는 아이디어를 발명해낸 것은 아니다. 이식성 Portability 은 컴퓨터 산업의 성배와 같은 것이다. 예를 들어 마이크로소프트는 원래 윈도우 NT가

인텔 머신에서도 운영되면서 동시에 워크스테이션 시장에서 흔히 볼 수 있는 RISC 머신에서도 운영될 수 있는 이식성 있는 운영체제가 되길 희망했다. 그러나 리눅스는 그러한 야망에 찬 목표를 처음부터 가져본 적이 없었다. 하지만 아이러니하게도 리눅스는 다중 플랫폼 코드의 성공적인 모델이 되었다.

원래 리눅스는 인텔 386 아키텍처 하나를 목표로 하였다. 오늘날 리눅스는 팜파일렛부터 알파 워크스테이션에 이르는 모든 플랫폼에서 동작하는, PC에 사용할 수 있는 운영체제 중 가장 폭넓게 이식된 운영체제다. 리눅스에서 작동하는 프로그램을 작성하면 그 프로그램이 바로 폭넓은 하드웨어에 대하여 “한 번 작성하여 모든 곳에서 실행한다.”가 된다. 그럼 리눅스가 어떻게 처음 예상과는 다른 것이 되었는지 이해하기 위해서 리눅스 설계에 어떤 결정 사항이 영향을 미쳤고, 어떻게 리눅스 개발 노력이 진전되었는지 살펴보자.

## 아미가와 모토로라 포트

리눅스는 유닉스와 비슷한 운영체제지만 유닉스 버전은 아니다. 예를 들어 리눅스는 FreeBSD와 다른 유산을 갖고 있다. FreeBSD의 창시자는 베클리 유닉스의 소스 코드를 가지고 시작했으며 그 커널은 베클리 유닉스 커널로부터 직접 나온 것이다. 따라서 FreeBSD는 유닉스의 한 버전이다. 즉, FreeBSD는 유닉스의 한 범주에 속한다. 한편 리눅스는 유닉스와 호환되는 인터페이스를 제공하는 것을 목표로 하고 있으나, 커널은 유닉스 소스 코드를 참고하지 않고 처음부터 새롭게 작성되었다. 따라서 리눅스 자체는 유닉스 포트<sup>port</sup> 중 하나가 아니라 새로운 운영체제다.

사실 처음부터 이 새로운 운영체제를 다른 플랫폼으로 이식하길 생각하지 않았으며, 단지 내가 갖고 있는 386에서 동작하는 무언가를 원했을 뿐이었다. 리눅스 커널 코드를 쉽게 이식할 수 있도록 만들기 위한 본격적인 노력은 리눅스를 DEC 알파 머신에 이식할 때 시작되었다. 그러나 알파 포트가 첫 번째 포트는 아니었다.

첫 번째 포트는 초기 썬, 애플, 아미가<sup>Amiga</sup> 컴퓨터에서 사용되고 있는 칩인 모토로라 68K 시리즈에 리눅스 커널을 이식한 팀에서 나왔다. 모토로라 포트를 했던 프로그래머들은 뭔가 저수준의 것을 하길 원했고, 유럽에는 아미가 공동체, 특히 DOS나 윈도우를 사용하기 싫어하는 층에 이런 사람들이 많았다.

아미가 사용자가 68K에서 동작하는 시스템을 만들어내기는 했지만 성공적인 리눅스 포트라고 생각하지는 않는다. 그들은 내가 처음 리눅스를 만들기 시작했을 때의 접근 방법을 사용하였다. 즉, 어떤 특정 인터페이스를 지원하기 위해 처음부터 코드를 작성한 것이다. 따라서 첫 번째 68K 포트는 리눅스와 유사한 운영체제라고 생각할 수 있으며, 원래의 코드에서 떨어져 나오는 포크<sup>fork</sup>(분리되어 나온 것)라고 할 수 있다.

한편으로는 이 첫 번째 68K 리눅스가 이식성이 뛰어난 리눅스를 만드는데 전혀 도움이 되지 않았다고 볼 수도 있지만, 어찌보면 도움이 되었다고도 볼 수 있다. 내가 알파 포트에 대하여 생각하기 시작했을 때 68K의 경험을 고려해야 했다.

만약 알파에 대해 똑같은 접근 방법을 취했다면 리눅스를 유지하기 위해 3개의 서로 다른 코드 기반을 가져야 했을 것이다. 이는 코딩의 관점에서는 가능할지는 몰라도 관리 관점에서는 전혀 그렇지 않다. 누군가 새로운 아키텍처에 리눅스를 원할 때마다 새로운 코드 기반을 유지해야 한다면 나는 리눅스 개발 관리를 해낼 수 없었을 것이다. 대신에 나는 알파 고유의 소스 구조와 68K 고유의 소스 구조, x86 고유의 소스 구조를 가지면서 모두 공통의 코드 기반을 갖는 그런 시스템을 원했다.

따라서 이때 커널 대부분을 새롭게 다시 작성해야 했는데, 이 코드 재작성은 늘어나는 개발자 공동체와 어떻게 일할 수 있는지를 염두에 두고 진행되었다.

## 마이크로커널

리눅스 커널을 만들기 시작했을 때, 최대의 관건은 어떻게 이식성이 뛰어난 시스템

을 만들 수 있는가였다. 여기에 대해 인정받던 학파가 있었는데 그들의 주장은 마이크로커널 스타일의 아키텍처를 사용해야 한다는 것이었다.

리눅스 커널과 같은 모노리딕<sup>monolithic</sup> 커널에서는 메모리가 커널 영역과 사용자 영역으로 나뉜다. 커널 영역은 실제 커널 코드가 적재되고 커널 수준의 연산에 대한 메모리가 할당되는 곳이다. 커널 연산에는 스케줄링, 프로세스 관리, 시그널링, 장치 입/출력, 페이징, 스와핑 등이 포함된다. 이러한 핵심 기능에 의해 프로그램이 동작한다. 커널 코드에 하드웨어와 관련된 저수준 상호작용이 포함되어 있기 때문에 모노리딕 커널은 특정 아키텍처에 묶여 있다고 보기 쉽다.

마이크로커널은 훨씬 적은 연산을 좀 더 제한적인 형태로 수행한다. 상호 프로세스 통신, 제한된 프로세스 관리, 스케줄링, 그리고 몇 가지 저수준 입/출력이 이에 속 한다. 마이크로커널에서는 많은 시스템 특정 부분이 사용자 영역으로 밀려나 있기 때문에 하드웨어 특성과는 다소 거리가 있어 보인다. 마이크로커널 아키텍처는 기본적으로 프로세스 제어, 메모리 할당, 자원 할당 등의 세부 사항을 추상화하여 다른 칩셋으로의 이식에서 최소의 변화만 필요하게 하자는 방식이다.

따라서 내가 1991년 리눅스에 대한 작업을 시작할 당시 사람들은 이식성이 마이크로커널 접근 방식으로부터 나올 수 있다고 생각하였다. 여러분도 알다시피 이 생각은 그 당시 컴퓨터 과학자들을 매료시켰던 그런 방식이었다. 하지만 나는 실용적인 측면에서 당시 마이크로커널이란 (a) 실험적이며, (b) 분명히 모노리딕 커널 방식 보다 훨씬 더 복잡하며, (c) 모노리딕 커널보다 훨씬 느리게 동작한다고 느꼈다. 운영체제는 속도가 매우 중요하다. 따라서 그 당시 정상적인 커널만큼 마이크로커널의 속도를 높이기 위해 많은 연구 자금이 마이크로커널의 최적화 작업에 사용되었다. 재미있는 사실은 여러분이 실제로 논문을 읽어 보면 연구자들이 마이크로커널에 사용했던 최적화 트릭들이 사실은 전통적인 커널에도 적용되어 실행 속도를 높일 수 있다는 점을 발견할 수 있다는 점이다.

사실 이 때문에 나는 마이크로커널 접근 방식이 근본적으로 연구를 위해 더 많은 자금을 받기 위한 정직하지 못한 방법이라고 느꼈다. 꼭 그 연구자가 특별히 더 정직하지 못하다고 생각하지는 않는다. 아마도 그냥 어리석거나 속은 것일 수 있다. 내가 지금 말하는 것은 현실적인 문제다. 이러한 정직하지 못함은 그 당시 마이크로커널이라는 주제를 추구하는 연구 공동체의 심한 압박감에서 비롯된다. 컴퓨터 과학 연구소에서 마이크로커널을 연구하든지 아니면 연구 자체를 그만두어야 했다. 따라서 모든 사람은 이러한 정직하지 못함으로 내몰렸고 이는 윈도우 NT를 설계하는 사람들도 마찬가지였다. NT 팀은 최종 결과가 마이크로커널이 되지 않을 것이라는 사실을 알면서도 마이크로커널이라는 아이디어에 대하여 찬사를 늘어놓는 립 서비스를 해야 한다는 사실을 알고 있었다.

다행히도 나는 마이크로커널을 추구해야 한다는 심한 압박감을 느끼지 않았다. 헬싱키 대학은 지난 1960년대부터 운영체제 연구를 해왔으며 헬싱키 대학 사람들은 헬싱키 대학의 운영체제 커널을 더 이상 연구 주제감으로 보지 않았다. 한편으로는 그들이 옳았다. 왜냐하면 운영체제의 기본과 확장인 리눅스 커널은 초기 70년대에 잘 이해되었기 때문이었다. 그 후의 모든 것은 어느 정도 자기 만족을 위한 연구였을 뿐이다.

이식성이 뛰어난 코드를 원한다고 해서 꼭 이식성을 이루는 추상 계층을 만들 필요는 없다. 다른 대안이 있으면 그 대안을 사용하면 된다. 근본적으로 마이크로커널의 이식성을 높이려는 노력은 시간 낭비다. 마치 사각형 타이어를 달고 있으면서도 빠른 자동차를 만들려는 것과 같다. 무조건 빨라야만 하는 것, 바로 그러한 커널의 추상화 노력은 본질적으로 비생산적이다.

물론 마이크로커널 연구는 그 이상의 것이었지만, 문제는 목표 상의 차이다. 마이크로커널 연구 대부분의 목표는 이론적인 이상형 설계면서 어떤 아키텍처에도 이식할 수 있는 설계를 갖는 것이다. 하지만 리눅스에서는 그렇게 지고한 목표를 갖

지 않았다. 나는 이론적 시스템이 아니라 실제로 사용되는 시스템 간 이식성에 관심을 갖고 있었다.

## 알파로부터 이식성까지

알파 포트는 1993년에 시작되었고 완성되기까지 약 1년여가 소요되었다. 물론 1년 후에도 완벽하지는 못했지만 기초는 마련되어 있었다. 처음 포트를 작성하는 것은 힘든 작업이었지만, 그 첫 번째 포트 작업으로 지금까지 리눅스가 따라온 설계 원칙이 마련되었고 다음 작업부터는 수월하게 일을 진행할 수 있게 되었다.

리눅스 커널이 어떤 아키텍처에나 이식할 수 있도록 작성된 것은 아니었다. 나는 목표로 하는 아키텍처가 기본을 잘 갖추고 기초적인 규칙만 잘 따른다면 리눅스가 그러한 모델을 지원할 수 있을 것이라고 생각했다. 예를 들어 메모리 관리는 각 시스템마다 전혀 다를 수가 있다. 나는 68K와 SPARC, Alpha, 그리고 Power PC의 메모리 관리 관련 설명서를 읽고 그들이 세부적인 사항에서 약간씩 차이가 있지만 페이징과 캐시 등은 전반적으로 유사하다는 사실을 알게 되었다. 덕분에 나는 이 아키텍처 모두에 공통된 리눅스 커널 메모리 관리 부분을 작성할 수 있었다. 사실 특정 아키텍처의 세부 사항에 적용할 수 있도록 코드를 수정하는 것은 그다지 어려운 일은 아니었다.

이식 Porting, 포팅이라는 문제는 몇 가지 가정을 통해 단순화할 수 있다. 예를 들어 CPU에 페이징을 해야 한다면 CPU가 사용하는 가상 메모리를 매핑할 수 있는 변환 참조 버퍼 Translation Lookup Buffer, TLB를 CPU 확장 기능으로 갖추어야 한다. 물론 우리는 TLB의 형태에 대해서 알 필요가 없으며, 단지 어떻게 TLB를 채우고 비우는가만 알면 된다. 따라서 우리는 정상적인 아키텍처에서 커널에 몇몇 특화된 부분을 갖추면 된다. 하지만 코드 대부분은 TLB의 작동 방식과 같은 일반적인 메커니즘에 의존한다.

내가 따르는 몇 가지 원칙 중 하나는 항상 변수보다는 컴파일 타임 상수를 사용하는 것이 더 낫다는 것으로, 이 원칙을 따르면 코드를 최적화하는 작업에 있어서 컴파일러가 훨씬 나은 결과를 도출해내는 것을 볼 수 있다. 이 방법을 통해 우리는 코드를 좀 더 유연하게 정의하면서도 쉽게 최적화할 수 있다.

공통 아키텍처를 정의하기 위한 이 접근 방식은 실제로 하드웨어 플랫폼에서 작동하는 것보다 더 나은 아키텍처를 운영체제에 제공한다는 점에서 매우 흥미롭다. 쉽게 이해할 수는 없겠지만 이 사실은 매우 중요하다. 사실 우리가 시스템을 조사할 때 찾으려 하는 일반화라는 것은 종종 커널의 성능을 향상하기 위해 시도하는 최적화와 동일한 의미로 사용되기도 한다.

여러분도 알다시피 당신이 페이지 테이블 구현과 같은 작업을 위해 광범위한 조사를 하고 그 조사와 관찰에 의거하여 결정을 내릴 경우(예를 들어 페이지 트리가 3 단계로만 되어야 한다면)에 뛰어난 성능을 갖도록 하는 데만 흥미를 가진다면 결국 그런 방식으로 할 수밖에 없다는 사실을 알게 될 것이다.

다시 말해 이식성을 설계상의 목표로 삼는 것이 아니라 단순히 특정한 아키텍처의 커널에 대한 최적화로만 생각해도 똑같은 결론에 도달하게 될 것이라는 뜻이다(페이지 트리가 나타내는 커널에 대한 최적화의 수준이 3단계라는 뜻이다). 이는 단지 운이 좋은 것이 아니다. 어떤 아키텍처가 세부 사항에서 정상적인 공통 설계를 벗어난다면 그 원인은 설계 자체가 잘못되었기 때문인 경우가 많다. 따라서 이식성을 높이기 위해 설계 상에 특수성을 부여하려 한다면, 동시에 좋지 못한 디자인을 감안하고 공통 설계를 좀더 최적화시키려는 노력을 해야 한다. 기본적으로 나는 오늘 날의 컴퓨터 아키텍처에 있어서 존재하는 현실에 최상의 이론을 결부시키기 위해 중간자적 입장을 고수하려 노력해왔다.

## 커널 영역과 사용자 영역

리눅스 커널과 같은 모노리티 커널에서 새로운 코드와 기능을 커널에 허용할 때는 매우 신중해야 한다. 이 결정은 나중에 핵심 커널 차원의 작업을 넘어서는 개발 사이클에서 많은 것에 영향을 줄 수 있기 때문이다.

첫 번째 가장 기본적인 규칙은 인터페이스를 피하는 것이다. 누군가 새로운 시스템 인터페이스를 포함하는 것을 추가하길 바란다면 여러분은 상당히 신중을 기해야 한다. 일단 사용자에게 인터페이스를 제공하면 사용자는 인터페이스를 가지고 코딩을 할 것이며, 누군가 코딩을 시작했다면 여러분은 꼼꼼 없이 그 인터페이스에 간접하게 된다. 평생동안 똑같은 인터페이스를 지원하고 싶은가?

다른 코드는 별로 문제가 되지 않는다. 예를 들어 디스크 드라이버처럼 인터페이스를 갖지 않은 코드라면 그리 걱정할 필요가 없으며 별다른 위험 없이 새로운 디스크 드라이버를 추가할 수 있다. 비록 리눅스가 디스크 드라이버를 갖지 않더라도 드라이버를 추가하는 것은 리눅스를 사용하는 기존의 사용자 누구에게도 해를 끼치지는 않으며, 새로운 사용자에게는 리눅스를 열어주는 것이 된다.

다른 것에 대해서는 균형을 유지해야 한다. 좋은 구현인가? 정말로 좋은 기능을 추가하고 있는 것인가? 어떤 때는 기능이 좋음에도 불구하고 인터페이스가 나쁘거나 그것 때문에 지금 또는 나중에 다른 것을 할 수 없게 될 수도 있다. 인터페이스 문제긴 하지만, 예를 들어 누군가의 이름 길이가 최대 14개의 문자인 이상한 파일 시스템을 구현했다고 하자. 여러분이 정말로 피하고 싶은 것은 어떤 제한 사항을 인터페이스에 고정시켜 버리는 것이다. 파일 시스템을 채택해버리면 이 파일 시스템을 확장하고자 할 때 이 제한된 인터페이스에 맞출 방법을 찾아야 하므로 난처한 상황에 빠질 것이다. 이보다 더 나쁜 결과로, 파일 이름을 요청한 모든 프로그램에서 13개의 문자만 저장할 수 있는 변수를 가진 상황에서 더 긴 파일 이름을 전달하게 되면 프로그램이 죽어버리게 될 것이다.

현재로서 이렇게 하는 유일한 업체는 마이크로소프트다. 기본적으로 DOS/윈도우 파일을 읽기 위해서는 모든 파일 이름이 8개의 문자와 추가 문자 3개를 갖는 웃긴 인터페이스를 가져야 한다. 긴 파일 이름을 지원하는 NT에서는 더 긴 파일 이름을 처리하는 것을 제외하고는 다른 루틴과 똑같은 일을 하는 완전히 새로운 루틴을 가져야 한다. 이는 미래의 작업에 장애가 되는 인터페이스의 한 예다.

또 다른 예가 Plan 9 운영체제에서 일어났다. 그들은 좀 더 나은 방법으로 프로세스 포크를 처리할 수 있는 훌륭한 시스템 콜을 가지고 있었다. 이는 프로그램이 자신을 2개로 분리하고 각각 프로세싱을 지속할 수 있도록 해주는 간단한 방법을 제공한다. Plan 9에서 R-Fork(그리고 나중에 SGI에서는 S-Proc이라 부르는)라고 부르는 새로운 포크는 기본적으로 주소 영역을 공유하는 2개의 프로세스 영역을 만들어 내며, 특히 스레드에 도움이 된다.

리눅스는 클론<sup>clone</sup> 시스템 콜을 통하여 동일한 일을 하지만 그들과 달리 올바르게 구현되어 있다. SGI와 Plan 9 루틴에서는 2개로 나눈 프로그램이 같은 주소 영역을 사용하지만 별도의 스택을 사용하도록 하였다. 정상적으로는 두 개의 스레드에서 같은 주소를 사용하면 같은 메모리 위치를 갖게 된다. 하지만 특별한 스택 세그먼트를 가지게 되며, 스택 기반의 메모리 영역을 사용하게 되면 실제로는 다른 스택을 덮어쓰지 않으면서도 스택 포인터를 공유할 수 있는 두 개의 서로 다른 메모리 위치를 얻게된다.

이는 지능적인 재주이긴 하지만 스택을 관리하는 데 드는 오버헤드 때문에 실제로는 없는 것만 못한 것이 되어 버린다. 그들은 뒤늦게 성능이 어마어마하게 떨어진다는 사실을 알게 되었다. 하지만 이미 이 인터페이스를 사용하는 프로그램들이 있었기 때문에 고칠 수 없게 되었다. 대신 그들은 스택 영역을 현명하게 관리할 수 있도록 적절하게 작성된 새로운 인터페이스를 추가하였다.

독점 업체에서는 종종 설계상 결함을 아키텍처에 밀어 넣는 시도를 할 수 있겠지만 리눅스의 경우에는 이런 식으로 하지 않는다.

리눅스 개발을 관리하고 리눅스에 대한 설계 결정을 내리는 데 있어 똑같은 접근 방식을 사용하도록 하는 또 다른 경우를 설명하고자 한다. 현실적으로 나는 커널에 인터페이스를 제공하는 많은 개발자를 관리할 수가 없었다. 아마도 커널에 대한 통제를 할 수 없기 때문이었을 것이다. 하지만 설계의 관점에서도 커널을 비교적 쉽게 유지하고 인터페이스의 개수와 앞으로의 개발을 제한할 요소가 있다면 이를 최소로 유지하는 것이 좋다.

물론 리눅스도 이런 면에서 완벽하게 깨끗하다고 할 수는 없다. 리눅스는 유닉스의 이전 구현으로부터 수많은 끔찍한 인터페이스를 물려받았다. 만약 리눅스가 유닉스로부터 물려받은 인터페이스를 유지할 필요가 없었다면 어떤 측면에서는 지금보다 훨씬 나았을지도 모른다. 하지만 리눅스는 완전히 처음부터 시작하는 시스템에서 기대할 수 있는 그런 깨끗함을 지니고 있다. 유닉스 애플리케이션을 실행할 수 있는 혜택을 바란다면 결과적으로는 유닉스라는 짐을 지게 된다. 리눅스가 널리 사용되기 위해서는 유닉스 애플리케이션을 실행할 수 있어야 한다는 사항이 필수적이므로 타협의 가치는 있다.

## GCC

유닉스 그 자체는 이식성의 관점에서 볼 때 대성공이다. 많은 커널과 마찬가지로 유닉스 커널도 이식성의 대부분을 C 언어에 의존하며 리눅스도 마찬가지다. 유닉스의 경우, 많은 아키텍처에서 C 컴파일러가 존재하기 때문에 유닉스를 이식하는 일이 가능해졌다. 따라서 유닉스는 컴파일러의 중요성을 상당히 강조하고 있으며, 리눅스의 라이선스를 GPL<sup>GNU Public License</sup>로 선택하는 데도 이 컴파일러의 중요성이 어느 정도 영향을 주었다. GPL은 GCC 컴파일러의 라이선스다.

내 생각으로는 GNU 그룹에서 나온 다른 프로젝트 모두는 상대적으로 리눅스에 중요하지 않다. 내가 유일하게 관심을 갖는 것은 바로 GCC다. 나는 GNU 프로그램의 상당수를 싫어한다. 예를 들어 Emacs는 끔찍할 정도다. 리눅스는 Emacs보다 거대하긴 하지만 최소한 그렇게 커야만 되는 명분을 가지고 있다. 하지만 기본적으로 컴파일러는 근본적인 필수품이다.

현재 리눅스 커널은 일반적으로 이식 가능한 설계를 따르고 있기 때문에 최소한 제대로 된 아키텍처에 대해 적당한 컴파일러만 있다면 이식할 수 있다. 앞으로 나올 칩에 대해서도 커널에 관한 한 아키텍처의 이식성에 대해서는 별로 걱정하지 않는다(나는 컴파일러에 대한 걱정을 한다). 인텔의 64비트 칩인 머세드<sup>Merced</sup>가 가장 좋은 예인데, 왜냐하면 컴파일러의 관점에서 보면 머세드는 매우 다른 칩이기 때문이다. 따라서 리눅스의 이식성은 GCC가 주요 칩 아키텍처에 이식되어 있다는 사실과 긴밀하게 연관된다.

## 커널 모듈

리눅스 커널에 대해 우리가 원하는 시스템은 최대한 모듈화<sup>module</sup>되어 있어야 한다는 사실은 분명하다. 오픈 소스 개발 모델에서는 정말로 모듈화가 필요하다. 왜냐하면 모듈화되지 않은 경우 사람들이 함께 작업하도록 하는 일이 쉽지 않기 때문이다. 많은 사람이 커널이라고 하는 똑같은 부분에 대하여 작업하고 충돌을 일으킨다면 정말 고통스러울 것이다.

모듈성이 없다면 다른 것에 영향을 미칠만한 변화 사항은 없는지 알아보기 위해 고쳐진 파일을 모두 점검해야 할 것이다. 하지만 모듈성을 가진다면 누군가가 보내온 새로운 파일 시스템을 구현하는 패치가 믿을 만한 구현이 아니더라도 이 파일 시스템을 사용하지 않으면 다른 것에 영향을 주지 않는다는 사실을 믿을 수 있다.

예를 들어, 한스 라이저<sup>Hans Reiser</sup>는 새로운 파일 시스템을 위해 작업하고 있으며

이제 제대로 작동하는 단계까지 와 있다. 이 시점에서 2.2 커널에 넣을 필요성은 느끼지 못하고 있다. 하지만 커널의 모듈성 덕분에 원하기만 한다면 넣을 수 있으며 그렇게 어려운 일도 아니다. 중요한 점은 그로 인해 다른 사람에게 피해가 가지 않도록 해야 한다는 것이다.

2.0 커널에서 리눅스는 정말로 크게 성장했다. 이때가 바로 적재 가능한 커널 모듈을 추가한 시점으로, 모듈을 작성할 수 있는 명시적인 구조를 만들어 모듈성을 명백히 향상시켰다. 덕분에 프로그래머 사이의 충돌이라는 부담 없이 서로 다른 모듈을 만들 수 있었고, 커널 안에 작성된 내용에 대한 제어권을 가질 수 있었다. 사람들과 코드를 관리하는 것 또한 똑같은 결론에 이르게 되었다. 많은 사람이 조화롭게 리눅스에서 작업할 수 있도록 유지하기 위해서는 커널 모듈과 같은 것이 필요했으며 설계 측면에서도 옳은 일이었다.

모듈성의 다른 부분은 좀 더 불분명하며 많은 문제점이 나타난다. 모든 사람이 좋다고 동의하는 것은 런타임<sup>run-time</sup> 로딩 부분이다. 하지만 새로운 문제를 낳는다.

첫 번째 문제는 기술적인 것이다. 하지만 기술적인 문제는 (거의) 항상 가장 쉬운 문제다. 더 중요한 문제는 기술과 상관 없는 문제다. 예를 들어 어느 시점에서 모듈이 리눅스로부터 유래한 작업이 되어 GPL 규약을 준수해야 하는가라는 문제가 있을 수 있다.

첫 번째 모듈 인터페이스가 만들어졌을 때 SCO용 드라이버를 만든 사람 중에는 GPL이 요구한 대로 소스 코드를 공개하지 않고 리눅스용 바이너리를 컴파일하여 제공하려는 사람들이 있었다. 그때 도덕적인 이유에서 이런 상황에 대하여 GPL을 적용하지 않겠다고 결정하였다.

GPL에서는 GPL을 라이선스로 갖는 작업에서 ‘유래한’ 작업 또한 GPL을 채택해야 한다고 요구한다. 불행히도 유래한 작업이라고 하는 것은 약간 모호하다. 유래

한 작업과 아닌 것 사이에 선을 그으려 할 때 도대체 어디에 선을 그어야할 것인가라는 문제가 발생한다.

우리는 결론적으로(또는 내가 선포했다고 할 수도 있다) 시스템 콜은 커널에 링크되는 것이 아니라고 보았다. 즉, 리눅스 위에서 동작하는 모든 프로그램이 GPL이어야 할 필요는 없다. 이 결정은 초창기에 이루어진 것이며 모든 사람이 알 수 있도록 특별한 README 파일을 추가하기도 했다(Vol. II 부록 참고). 이 덕분에 업체는 GPL에 대하여 걱정하지 않고도 리눅스용 프로그램을 작성할 수 있다.

결론적으로 모듈 제작자는 적재를 위해 정상적인 인터페이스만 사용한다면 폐쇄적인 모듈을 작성할 수 있다. 물론 이것은 여전히 커널의 어색한 영역이다. 이 중간 영역은 사람들이 무엇인가 이용할 수 있는 협점을 남기고 있으며, 이는 부분적으로 GPL이 모듈 인터페이스와 같은 것에 대하여 명확하게 하지 않음이기도 하다.

만약 누군가 단지 GPL을 우회하기 위해 엑스포트된 심볼<sup>exported symbol</sup>을 사용하여 우리가 제시한 가이드라인을 악용한다면 내 생각으로는 그 사람을 고소해야 할 것이다. 하지만 어느 누구도 커널을 악용하려고 한다고 생각하지 않는다. 커널에 상업적인 관심을 보인 사람들이 그렇게 한 이유는 리눅스 개발 모델에서 얻을 수 있는 혜택에 관심을 갖고 있었기 때문이다.

리눅스의 힘은 코드 자체만큼이나 그 뒤에서 협력하는 공동체의 힘에 기반을 둔다. 만약 리눅스를 탈취하여 누군가 독점적인 버전을 만들려고 한다면 그 독점적인 버전에서는 근본적으로 오픈 소스 개발 모델인 리눅스의 매력이 사라질 것이다.

## 오늘날의 이식성

오늘날의 리눅스는 사람들이 오로지 마이크로커널만이 이를 수 있을 것이라고 생각했던 많은 설계 목표를 이룩하였다. 전형적인 아키텍처 사이의 공통적인 요소들로부터 일반적인 커널 모델을 만들어 냄으로써 리눅스 커널은 마이크로커널처럼

성능면에서 불이익을 감수하지 않고도 추상 계층 없이 이식성이 많은 이점을 가지게 되었다. 커널 모듈을 사용함으로써 하드웨어 고유의 코드를 종종 모듈에 제한시키고 핵심 커널 부분의 이식성을 유지하였다. 장치 드라이버는 모듈 안에 하드웨어 고유의 코드를 제한시키기 위해 커널 모듈을 효과적으로 사용하는 좋은 예다. 이는 하드웨어의 고유한 부분을 핵심 커널에 넣음으로써 빠르지만 이식성이 없는 커널을 만드는 것과 모든 하드웨어 고유 부분을 사용자 영역에 넣어 느리거나 불안정적인 시스템을 만드는 것 사이에 놓인 훌륭한 중간 영역이다.

하지만 이식성에 대하여 리눅스가 취한 접근 방식은 리눅스를 둘러싸고 있는 공동체에게도 좋은 것이었다. 이식성이 동기가 된 결정을 통해 커널이 내 통제권을 벗어나지 않으면서도 많은 그룹이 동시에 리눅스의 각 부분에 대하여 작업할 수 있게 되었다. 리눅스가 기초를 두는 아키텍처 일반화를 통해 나는 커널 변화를 점검할 수 있는 참조 틀을 갖게 되었고, 각 아키텍처에 대하여 개별적인 코드 포크를 하지 않아도 되도록 추상화할 수 있었다. 따라서 많은 사람이 리눅스에 대하여 작업하지만 핵심 커널은 내가 유지할 수 있는 상태로 유지되고 있다. 그리고 커널 모듈을 통해 프로그래머는 서로 독립적이어야 하는 시스템의 일부에 대하여 정말로 독립적인 작업을 할 수 있는 명확한 방법을 갖게 되었다.

## 리눅스의 미래

나는 커널 영역에서 가능한 한 최소한의 것을 함으로써 우리가 리눅스에 대하여 옳은 결정을 내렸다고 확신한다. 여기서 솔직한 진실을 말하자면 나는 커널에 가해지는 주요 간섭 사항을 미리 생각하지 않는다. 커널에서 중요한 혁신은 그리 많지 않다. 무엇보다도 좀 더 폭넓은 시스템을 지원하는 것이 문제다. 즉 리눅스에 이식성을 활용하여 새로운 시스템에서 리눅스가 작동할 수 있도록 하는 것이다.

새로운 인터페이스가 생길 것이다. 하지만 내 생각으로는 새로운 인터페이스가 부분적으로 폭넓은 시스템을 지원하면서 생겨날 것이라고 본다. 예를 들어 클러스터

링을 하게 되면 여러분은 갑자기 스케줄러에게 특정 그룹의 프로세스에 대하여 집단 스케줄링<sup>gang scheduling</sup>과 같은 것을 하도록 말하고 싶을 것이다. 하지만 동시에 나는 모두가 클러스터링이나 슈퍼컴퓨팅에 초점을 맞추길 원하지 않는다. 왜냐하면 우리의 미래는 랩탑 컴퓨터 또는 여러분이 어디를 가든 꽂아 사용할 수 있는 카드 또는 그와 유사한 것에 있다고 생각하기 때문이다. 따라서 나는 리눅스가 그러한 방향으로도 나아가길 바란다.

그리고 정말로 사용자 인터페이스가 전혀 없는 임베디드 시스템<sup>embedded system</sup>이 있다. 아마도 커널을 업그레이드할 경우에만 시스템에 접근할 뿐, 다른 경우에는 그냥 놓여 있는 시스템일 것이다. 따라서 이는 리눅스의 또 다른 방향이 될 수 있다. 나는 자바나 인페노<sup>Inferno</sup>(루슨트의 내장 시스템용 운영체제)가 내장 장치에서 성공할 것이라고 생각하지 않는다. 그들은 무어의 법칙<sup>Moore's Law</sup>이 가진 중요성을 망각하였다. 맨 처음에는 특정 내장 장치에 최적화된 시스템을 설계하는 것이 그럴듯하게 들릴지는 모르나 작동 가능한 설계를 가질 때쯤에는 무어의 법칙에 의거하여 좀 더 강력한 하드웨어의 가격이 낮아짐으로써 특정 장치에 적합한 설계의 가치를 떨어뜨리게 될 것이다. 모든 것의 가격이 하락하여 같은 시스템을 여러분의 데스크톱과 내장 장치에 함께 가질 수 있게 될 것이다. 이렇게 되면 모든 사람의 생활이 더욱 편리해질 것이다.

대칭형 다중 프로세싱<sup>SMP</sup>은 개발되는 영역 중 하나다. 2.2 커널은 4개의 프로세서를 잘 처리할 것이며, 8개 또는 16개의 프로세서를 지원하기 위해 개발을 지속해 나갈 것이다. 4개 이상의 프로세서 지원은 이미 있지만 현실적인 것은 아니다. 현재 4개 이상의 프로세서를 가지고 있다면 죽은 말에 돈을 거는 것과 같다. 따라서 이 부분에 대한 향상이 이루어질 것이다. 하지만 사람들이 64개의 프로세서를 원한다면 특별한 버전의 커널을 사용해야 할 것이다. 왜냐하면 정상 커널에서 지원하려면 일반 사용자에게는 성능 상의 저하가 있기 때문이다.

몇몇 특정 애플리케이션 영역이 커널 개발을 지속해서 이끌 것이다. 웹 서비스는 정말로 커널 집약적인 진정한 애플리케이션이므로 언제나 흥미로운 문제였다. 한 편 웹 서비스는 내게 위험한 영역이다. 왜냐하면 리눅스를 웹 서비스 플랫폼으로 사용하는 사람들의 피드백을 많이 받음으로써 단지 웹 서비스에 대해서만 최적화 하는 데 그칠지도 모르기 때문이다. 웹 서비스 영역이 중요하기는 하지만 전부는 아니라는 사실을 염두에 두어야 한다.

물론 현재의 웹 서버에서도 리눅스의 모든 잠재 능력까지 사용하지는 못하고 있다. 예를 들어 아파치 그 자체는 스레드에 관련하여 제대로 하지 못하고 있다. 이런 종류의 최적화는 네트워크 대역폭 제한에 의해 개발 속도가 늦춰져 왔다. 현재 10메가비트 네트워크를 포화 상태로 만들기 때문에 더 이상 최적화할 이유가 없다. 10메가비트 네트워크를 포화시키지 않는 유일한 방법은 부하를 주는 수많은 CGI를 두는 것이다. 하지만 이 부분에 대하여 커널이 도울 수 있는 것은 없다. 커널이 할 수 있는 일이라면 정적 페이지에 대한 요청에 대해서는 직접 답변하고 좀 더 복잡한 요청은 아파치에게 전달하는 것이다. 일단 더 빠른 네트워크가 일반화되면 앞서 말한 내용은 점점 더 매력적인 것이 될 것이다. 그러나 현재로서는 테스트하고 개발할 만한 하드웨어가 별로 없다.

모든 가능한 미래의 방향으로부터 얻은 교훈은 리눅스가 최첨단 아니 최첨단을 조금 더 지나간 곳에 위치하길 바란다는 것이다. 왜냐하면 오늘날 첨단을 지난 것이 바로 내일 여러분의 데스크톱에 놓일 것이기 때문이다. 하지만 리눅스에 있어 가장 흥미로운 개발은 커널 영역이 아닌 사용자 영역에서 일어날 것이다. 커널의 변화는 앞으로 일어날 일과 비교할 때 상대적으로 작게 보일 것이다. 이런 관점에서 리눅스 커널은 레드햇 17.5에 어떤 기능이 있을 것인가 또는 윈도우 에뮬레이터 WINE가 몇 년 안에 어떻게 될 것인가의 문제만큼 흥미롭지는 않다.

15년 후 누군가 나와서 “여보시오. 나는 리눅스가 할 수 있는 모든 일을 할 수 있소. 그러나 내 시스템에는 짊어져야 할 20년 간의 짐이 없기 때문에 더 가볍고 간단하게 해낼 수 있소.”라고 말할 수 있는 날이 오기를 기대한다. 그들은 리눅스가 386용으로 설계되었고 새로운 CPU가 정말로 흥미로운 일을 전혀 다르게 한다고 말할 것이다. “이 오래된 리눅스를 버립시다.” 이것이 바로 내가 리눅스를 만들 때 한 일이다. 그리고 앞으로 그들이 우리의 코드를 살펴보고 인터페이스를 사용하며 바이너리 호환성을 제공할 수 있을 것이며, 이런 모든 일이 일어난다면 나는 행복할 것이다.