Hanbit eBook

Realtime 37

오픈 소스 Vol I

오픈 소스 혁명의 목소리

에릭 레이먼드, 리처드 스톨만, 리누스 토발스, 팀 오라일리 외 지음

송창훈, 이기동, 이만용, 최준호 옮김

한빛미디어 Hanbit Media, Inc.

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** 김태헌
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** 이중민
디자인 표지 여동일, 내지 스튜디오 [밈]
마케팅 박상용, 박주훈

이 책에 대한 의견이나 오탈자 및 잘못된 내용에 대한 수정 정보는 한빛미디어(주)의 홈페이지나 아래 이메일로 알려주십시오.
한빛미디어 홈페이지 www.hanb.co.kr / **이메일** ask@hanb.co.kr

Published by HANBIT Media, Inc. Printed in Korea

지금 하지 않으면 할 수 없는 일이 있습니다.
책으로 펴내고 싶은 아이디어나 원고를 메일(ebookwriter@hanb.co.kr)로 보내주세요.
한빛미디어(주)는 여러분의 소중한 경험과 지식을 기다리고 있습니다.

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

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

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 리누스, 래리 월을 포함한 프로그래머의 기술적인 입장과 리처드 스톨만, 에릭 레이먼드를 포함한 정신적, 이론적 리더의 주장, 그리고 밥 영, 팀 오라일리를 포함한 오픈 소스 벤처 사업가의 시장 가치의 관점 등 이 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 닐Neil, 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 헌신적인 사람들을 본 적이 없다. 내가 글을 쓰는 동안 참고 도와준 그들에게 감사하는 바다.

이 책은 마크 스톤Mark Stone의 지속적인 도움과 헌신으로 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 녹스Jeff Knox, 베키 우드Becky Wood와 http://slashdot.org의 촉매 역할을 했던 롭 말다Rob Malda 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 헌신이 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 디보나**ChrisDiBona

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

— **샘 오크만**Sam 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 시간보다는 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 코터리Nancy Wolfe Kotary, 셰릴Sheryl과 마이크 시에라Mike Sierra, 그리고 에디 프리만Edie 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 스톤**Mark 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 엄두를 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.

차례

서문

크리스 디보나Chris DiBona, 샘 오크만Sam Ockman, 마크 스톤Mark Stone

이만용 역

프롤로그

리눅스 창시자인 리누스 토발스Linus Torvalds에 의하면 리누스라는 이름은 부모님께서 노벨상 수상자인 리누스 폴링Linus Pauling을 존경했기 때문에 지어졌다고 한다. 폴링은 한 번도 아니고 두 번씩이나 노벨상을 받은, 흔치 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 크릭Fancis 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 Delbruk이 있다. 결과를 확신하기 전까지

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

> 먼저 리누스 폴링은 맥스 델브룩에게 이중 나선 구조에 대한 이야기를 들었다. 나는 보조 사슬에 대한 소식을 전하는 편지 말미에 폴링에게 이야기하지 말아 달라고 델브룩에게 요청했다. 아직은 무언가 잘못되지 않을까 하는 미심쩍은 상태였기 때문에 우리의 입장을 정리하기 위한 며칠의 여유를 갖기 위해서였다. 그때까지는 폴링이 수소가 결합한 기본 쌍에 대해 생각하지 않기를 바랬다. 하지만 내 요청은 무시당했다. 델브룩은 그의 연구소에 있는 모든 사람에게 이야기하기를 원했다. 몇 시간 만에 그 이야기는 그의 생물학 연구소로부터 리누스 밑에서 일하는 다른 친구들의 귀로 들어가게 되었다. 또한 델브룩은 나에게 전해 들은 소식을 폴링에게 전해 줄 것을 약속했었다. 그리고 무엇보다도 중요한 사실이 있다. 델브룩은 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 연구소의 연구원이었던 리처드 스톨만Richard Stallman은 GNU 프로젝트를 시작하였다. GNU 프로젝트의 목적은 간단히 말해 누구도 소프트웨어에 비용을 지불하지 않도록 하는 것이었다. 스톨만은 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 스톨만은 널리 공유하고 배포해야 하는 것으로 본다. GNU 프로젝트와 자유 소프트웨어 재단Free Software Foundation, FS F—GNU 프로젝트의 우산 조직이다—의 기본 신조는 소스 코드는 컴퓨터 과학을 발전시키는 데 기본이며, 변화가 계속되기 위해 정말로 필요한 것은 바로 자유롭게 사용할 수 있는 소스 코드라는 것이다.

스톨만은 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 스톨만은 GPL이라고 알려진 GNU 일반 공공 라이선스GNU General Public License로 대처하였다(부록 B 참고).

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.

스톨만을 비롯한 이 책의 저자들이 자유 소프트웨어에 대하여 말할 때는 사실 언론의 자유free speech에 대하여 말하는 것이다. 영어에서는 이 차이점을 제대로 구분하지 못하지만 "Free as in speech, not as in beer"에서처럼 무료gratis와 자유liberty는 구분되어야 한다. 이러한 급진적인 메시지(맥주 부분이 아니라 자유 부분의) 때문에 많은 소프트웨어 회사는 자유 소프트웨어를 거부하고 있다. 회사들이란 지식의 체계에 이바지하기 보다는 결국 돈을 버는 비즈니스 영역에 있기 때문이다. 추측하건대 스톨만은 컴퓨터 산업과 컴퓨터 과학 사이의 분열을 당연한 것으로 받아들이며 심지어는 원한다고도 할 수 있다.

오픈 소스 소프트웨어란 무엇인가?

1997년 봄, 자유 소프트웨어 공동체의 리더 그룹들이 캘리포니아에서 회합을 했다. 이 그룹에는 에릭 레이먼드Eric Raymond, 팀 오라일리Tim O'Reilly, VA 리서치사 사장인 래리 오거스틴Larry Augustin이 포함되어 있었다. 그들의 관심사는 기존 자유 소프트웨어라는 개념에 대해 거부감을 가진 사람들에게 자유 소프트웨어를 둘러싼 아이디어들을 진흥시킬 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 진흥시키고자 하는 소프트웨어를 나타내는 새로운 용어인 오픈 소스Open Source가 나왔다. 그리고 오픈 소스라고 말할 수 있는 소프트웨어에 대한 가이드라인을 만들었다.

브루스 페런스Bruce 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 크레딧credit없이도 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[01]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 페니실린의 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 인터페이스GUI 모두 제록스Xerox PARC에서 나왔다. 그러나 컴퓨터 산업에 불길한 징조가 없는 것은 아니다. 레드몬드Redmond(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 버클리의 두 학생이 졸업하면서 산업체로 직장을 얻으면서 그들의 기술 혁신을 제쳐 놓았기 때문이다.

소스를 사용하거라! 루크02

오픈 소스란 위로부터 결정된 아이디어가 아니다. 오픈 소스 운동은 진정한 풀뿌리 혁명이다. 에릭 레이먼드와 브루스 페런스와 같은 전도사가 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 마음에서라기보다는 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 영Bob Young의 회사인 레드햇Red Hat 소프트웨어사는 자사의 핵심 제품인 레드햇 리눅스를 무료로 배포함으로써 번창하고 있다. 자유 소프트웨어를 전달하는 효과적인 방법의 하나는 훌륭한 매뉴얼과 함께 완전한 기능을 갖춘 배포판으로 패키지하는 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. 칩바이트Cheap Bytes와 리눅스 센트럴Linux 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 무료로도 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 Allmen은 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"과 같은 주소를 실제 IP 주소(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 기술혁신Innovation이다.

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 60년대에 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 개발 과정 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 유저end-user 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 T나 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 서버IIS 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 보잘것없어 보인다.

하지만 아이러니는 심화되고 있다. QUESO와 WTF에 의한 조사 자료에 의하면 웹의 29%가 리눅스 기반의 서버로 운영되고 있다. QUESO는 TCP/IP 패킷을 보내 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 코히런트Coherent는 이미 파산하였다. 산타크루즈 오퍼레이션SantaCruz 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 스팍SPARC 이식을 돕기 위해 하드웨어와 자원을 기증하고 존 아우스터하우트John Ousterhout의 Tcl 스크립트 언어 개발을 지원하는 등, 수년간 여러모로 오픈 소스 개발을 지원해왔다. 커크 맥퀴식Kirk McKusick이 말하듯 버클리 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 창출한다. 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 이룰 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 번번스러운 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.[03]

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 토발스가 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 서드 파티 드라이버를 포함하고 GUI 부분을 커널 일부로 만든 점을 고치려면 마이크로소프트는 어마어마하게 느린 에뮬레이션 계층을 만들거나

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 잃을지 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 못할 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. 리누스 토발스, 앨런 콕스Alan Cox, 그리고 나머지 몇몇 사람이 탄탄하게 운영하고 있으며, 커널에 기능을 추가하고 접근하는 중앙 집권적 권위를 가지고 있다. 리눅스 커널 프로젝트는 리누스라는 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 확장할 수 있다. 그러나 소프트웨어 벤처는 마치 과학적 이론이 그러하듯 경쟁 과정 중에 빈번히 충돌한다. 루시드Lucid의 역사를 보자. 루시드는 인기 있는 프로그래머용 편집기인 EmacsEmacs를 활용하여 현대적인 버전을 만들고 리처드 스톨만이 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 가져다준다. 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 레이몬드의 fetchmail, 래리 월Larry Wall의 펄Perl, 그리고 리누스 토발스의 리눅스가 이런 식으로 시작되었다. 여러모로 자아성취는 스톨만이 GNU 프로젝트를 시작하게 한 동기의 기초를 이루는 중요한 개념이다.

리눅스 벤처와 미래 투자

해커의 동기는 지적이겠지만 꼭 그 결과가 희생을 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 몇천만 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, 스크립틱스Scriptics, 센드메일Sendmail, 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 스톨만은 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. 스코트 브래드너Scott Bradner가 말하는 바로는, 인터넷은 회사의 자본력이 아니라 참여하는 개개인 모두를 위해 유지되는 개방 표준의 강력한 집합 위에 건설되어 있다. 여러모로 인터넷은 원래 오픈 소스 벤처다. 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 관리자RPM를 패키지 관리 도구로 사용한다. 데비안Debian은 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 오라일리Tim O'Reilly가 묘사하는 인포웨어inforware 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. 왓슨과 크릭이 이중 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 뉴턴Newton이 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 명예reputation 게임에 관여한다. 그리고 역사는 과학적인 성공이 물질적인 성공보다 지속된다는 것을 보여주었다. 지난 백 년에 걸쳐 살아온 기업가 중 우리가 기억하는 사람이라곤 카네기Carnegie, 록펠러Rockefeller 등 극소수다. 반면에 우리는 지난 백 년에 걸쳐 살아온 수많은 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 스톨만이나 리누스 토발스와 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 쏟고 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 다스베이더(Darth Vader)의 아들이다. 다스베이더도 가르쳤던 제다이(Jedi) 기사 오비완(Obiwan)이 루크에게 포스(the Force)를 사용하라고 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 테크놀러지는 전 세계의 총명하고 창조적인 수많은 사람을 유혹하였다. 엑커트Eckert와 모츨리Mauchly의 에니악ENIAC에서부터 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 배치batch 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 Dictiondary', McGraw-Hill, 1981)'의 저자인 스탠 켈리 부틀Stan Kelly-Bootle과 뛰어난 전승자들은 1948년 최초의 동작하는 저장 프로그램 디지털 컴퓨터인 맨체스터 마크 IManchester Mark I에서 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. Lundstorm은 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 연구소Stanford Artificial Intelligence Laboratory, SAIL와 카네기 멜론 대학CMU도 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 버그도 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.[01]

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 알토 연구소Palo 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 네트워크LAN도 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 돕는 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 젊은 사람을 컴퓨팅 분야에 끌어들이기 위해 들여야 할 약간의 비용이라는 점을 이해하고 있었다.

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 건재하고 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 퍼졌고, 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 어셈블리어로 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 멍청하게 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 어셈블러를 사용하지 않았다. 반면에 C 프로그램은 이런 모든 기계 사이에서 즉시 사용할 수 있었다.

유닉스는 유닉스 간 복사 프로토콜Unix-to-Unix Copy Protocol, UUCP이라는 일종의 자신

만의 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 마이크로컴퓨터광의 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 버클리 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 스톨만Richard M. Stallman(Emacs 개발자)이었다.

스톨만(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.

스톨만의 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 것이었다. 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 윙 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 맥Mac 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 만들어지지 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 토발스Linus 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 리누스가 완전히 자유롭고 재배포 가능한 소스로 구성된 제대로 된 유닉스인 리눅스를 개발하도록 도왔다.

리눅스에게는 경쟁자가 없지 않았다. 1991년 리누스 토발스의 초기 실험과 동시에 윌리엄William과 라인 졸리츠Lynne Jolitz는 실험적으로 386에 BSD 유닉스 소스를 이식하였다. BSD 기술과 리누스의 미숙한 초기 노력을 비교하던 대부분의 관찰자는 BSD 포트port가 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 프리웨어

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, 다윈주의적인 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 ISPInternet Service Providers를 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 품위법Communications Decency Act, CDA'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

---

01 http://www.tuxedo.org 참고

2 | 버클리 유닉스의 20년

AT&T 소유에서 자유로운 재배포가 가능하기까지

마샬 커크 맥퀴식Marshall Kirk McKusick

최준호 역

초기 역사

켄 톰슨Ken Thompson과 데니스 리치Dennis Ritchie는 1973년 11월에 퍼듀Purdue 대학에서 있었던 SOSPSymposium of Operating Systems Principles에서 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 충돌은 문서에 있는 것과는 달리 디스크 컨트롤러가 중첩된 탐색을 신뢰성 있게 수행하지 못했기 때문에 일어났다. 톰슨이 보게 된 버클리의 11/45는 같은 컨트롤러에 두 개의 디스크가 달린 최초의 시스템 중 하나였다. 톰슨의 원격 디버깅은 버클리와 벨 연구소 사이에 이루어진 최초의 협동 작업이었다. 버클리와 작업을 공유하려 했던 벨 연구소 연구원들의 자발성은 버클리에서 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 웡Eugene 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 버전 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 버클리의 명성을 확립하는 데 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 보내기로 하였다. 톰슨은 제프 스크리브만Jeff Schriebman, 그리고 밥 크리들Bob Kridle과 함께 새롭게 설치된 11/70에 최신의 유닉스인 버전 6을 이식하였다.

1975년 가을 거의 남의 눈에 띄지 않는 대학원생이었던 빌 조이Bill Joy와 척 헬리Chuck 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 매리Queen Mary 대학의 조지 쿨루리스George 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 사항fifty 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 배포판Berkeley Software Distribution'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 Kessler와 내가 파스칼 시스템을 이식하고, 조이는 ex, vi, C 셸, 2BSD 배포판의 작고 수많은 유틸리티를 이식하였다. 1979년 말 완전한 배포판이 구성되었다. 이 배포판은 가상 메모리 커널, 표준 32/V 유틸리티, 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 야전 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 DARPADefense 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 DARPA와 1980년 4월에 시작하는 18개월짜리 계약을 할 수 있었다. 이 계약은 DARPA 계약자들이 필요한 기능을 추가하는 것이었다. 이 계약의 후원으로 밥 패브리는 CSRGComputer Systems Research Group, 컴퓨터 시스템 연구회라는 기구를 만들었다. 그는 곧 로라 통Laura 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 통은 조이의 이전 배포판보다 더 많은 양의 주문을 다룰 수 있는 배포 시스템을 만들었다. 패브리는 AT&T의 밥 거피Bob Guffy와 버

클리 대학의 변호사 모두가 납득할 수 있는 조항 아래에서 유닉스를 공식적으로 릴리즈하도록 조정하였다. 조이는 짐 컬프Jim Kulp의 작업 제어job 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 연구소Stanford Research Institute의 데이비드 카쉬탄David 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 엘즈Robert 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 기능Inter Process Communication, IPC을 제공하고, 네트워킹 지원을 통합하여 새 시스템을 운영하는 기계가 쉽게 ARPAnet에 참여할 수 있도록 할 예정이었다.

새 시스템을 규정하는 작업을 돕기 위해 DARPA에 있는 버클리와의 계약 감독관인 듀에인 아담스Duane Adams는 설계 작업을 돕고 연구자들의 요구가 정리되도록 '운영 위원회'로 알려진 그룹을 조직하였다. 이 위원회는 1981년 4월에서 1983년 6월까지 1년에 두 번 회의를 하였다. 여기에 속한 사람은 버클리 대학교의 밥 패브리, 빌 조이, 그리고 샘 레플러와 BBNBolt Beranek and Newman의 앨런 네메스, 롭 구르위츠, 벨 연구소의 데니스 리치, 스탠퍼드 대학의 키스 란츠, 카네기 멜론 대학의 릭 라쉬드, MIT의 버트 할스테드, ISIInformation Science Institute의 댄 린치, DARPA의 듀에인 애덤스와 밥 베이커, UCLA의 제리 폽펙이었다. 1984년부터 이 회의는 워크숍으로 바뀌어 더 많은 사람을 수용하게 되었다.

새 시스템에 포함될 기능을 제안하는 첫 번째 문서는 1981년 7월 운영 위원회와 버클리 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 구르위츠Rob 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 닥칠 시스템에 대한 큰 변화 때문에 교내 배포판도 피하는 것이 좋겠다고 느꼈는데, 특히 4.1a에서 4.1b로 전환하기 위해서는 모든 기계의 파일 시스템을 덤프하고 되돌려 놓아야 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 붙잡아 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 버클리 소프트웨어 배포판을 합친 것보다 더 많이 발송된 것이다. 대부분의 유닉스 벤더는 AT&T의 상용 시스템 V보다 4.2BSD 시스템을 판매하였다. 그 이유는 시스템 V에는 네트워킹도 없고 버클리 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 버클리는 인터페이스를 구현해야 한다고 DARPA에게 호소하였다. 따라서 버클리는 4.3BSD의 TCP/IP 코드를 BBN의 구현으로 교체해야 한다는 것이었다.

마이크 카렐스는 BBN 코드를 얻어 버클리에 원형이 제출된 이후의 작업에 대한 평가를 시작하였다. 그는 가장 좋은 방법은 버클리의 코드 베이스를 교체하지 않고 BBN 코드의 장점만을 버클리 코드 베이스에 통합하는 것이라고 결정하였다. 버클

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 절충안을 제시하였다.

마이크 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[Ballistics Research Laboratory]의 마이크 뮤스[Mike Muuse]에게 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 그러했듯이 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 보스틱[Keith Bostic]이 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 보스틱이 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 보스틱이 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 불렀다. 불행히도 이 아키텍처는 회사가 전략적 방향을 바꾸기로 결정함에 따라 없어지게 되었다. 그러나 CSRG에는 몇 개의 기계를 제공하여 빌 조이가 시작하였던 BSD 커널을 기계 의존적인 부분과 비의존적인 부분으로 분리하는 작업을 끝낼 수 있었다. 이 작업의 결과는 1988년 6월 4.3BSD-Tahoe로 릴리즈되었다. Tahoe라는 이름은 CCI사에서 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 버클리가 바이너리만의 형태로 BSD 시스템을 릴리즈한 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 버클리가 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 버클리와 그 공헌자들에 의해 개발된 것이었다. BSD에서 나온 네트워킹 코드와 지원 유틸리티는 네트워킹 릴리즈 1로 1989년 6월에 릴리즈되었으며, 이는 버클리 최초로 자유롭게 재배포할 수 있는 코드였다.

라이선스 조항은 자유로웠다. 라이선스를 받는 사람은 버클리와의 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 버클리 대학교와 공헌자들의 코드를 포함하고 있다고 알리는 것이었다. 버클리는 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 버클리에 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 통합하였

다(이는 또한 SunOS에서 사용한 인터페이스였다).

당시 시스템에 추가된 또 다른 중요한 사항은 썬과 호환되는 버전의 네트워크 파일 시스템NFS이었다. 또다시 CSRG는 실제 NFS 코드를 작성하지 않고 캐나다 구엘프Geulph 대학의 릭 맥클렘Rick Macklem이 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 유틸리티와 C 라이브러리의 재구현을 어떻게 다룰 것인지를 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 유틸리티를 다시 작성해 달라고 요청하였다. 보상은 오직 다시 작성된 유틸리티의 이름 뒤의 버클리 공헌자 속에 자신의 이름이 들어가는 것이었다. 공헌은 천천히 시작되었으며 대부분 사소한 유틸리티에 대한 것이었다. 그러나 완성된 유틸리티의 수가

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

보스틱은 자랑스럽게 목록을 손에 들고 마이크 카렐스와 나의 사무실 로 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 검증이 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 버클리에게서 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.org에서 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 컴덱스 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 에뮬레이션 모드를 추가하여 리눅스의 인기에 편승하였다. 이 기능은 FreeBSD 사용자가 FreeBSD 시스템의 튼튼함, 신뢰성, 성능과 함께 지속해

서 늘어가는 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.org에서 찾을 수 있다.

소송

네트워킹 릴리즈 2 테이프를 기반으로 설계된 시스템을 자유롭게 재배포하기 위한 그룹의 형성과 더불어 버클리 소프트웨어 디자인사Berkeley Software Design, Inc., 약칭 BSDI가 이 코드를 상업적으로 지원하는 버전을 개발하고 배포하려고 설립되었다(BSDI에 대한 더 많은 정보는 http://www.BSDI.com에서 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 보스틱과 나는 개인적으로 여러 가지 반증에 도움이 될 수백 페이지의 자료를 작성해야 했다.

1992년 12월, 미합중국 뉴저지 지방 법원의 디킨슨 데베브와즈Dickinson 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 버클리 대학은 다음 월요일 아침 USL에 대해 맞대항하는 고소장을 캘리포니아 주 법원에 제출하였다. 캘리포니아에서 먼저 제기됨에 따라 버클리는 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.

버클리 대학은 고소장에서 USL이 버클리 대학과 맺은 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 버클리 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 누어다Ray 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 버클리는 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 공헌자를 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

스코트 브래드너Scott Bradner

송창훈 역

IETFInternet Engineering Task Force [01]는 실재하지 않는 그 무엇을 이루는 데 큰 기여를 해왔다. TCP/IP 자체는 논외로 하더라도 IETF는 인터넷의 기반이 되는 모든 기술을 개발하고 개선해 왔다. IETF 워킹 그룹은 라우팅과 제어 방법 그리고 전송 규칙에 대한 표준을 만들었는데 이러한 표준이 없었다면 인터넷은 존재하지 못했을 것이다. 또한 IETF 워킹 그룹은 인터넷의 안전을 도모할 수 있는 보안 표준과 인터넷을 좀 더 예측 가능한 환경으로 만들기 위한 서비스 품질에 관한 표준, 그리고 차세대 인터넷 프로토콜 그 자체를 위한 표준들을 정의해왔다.

이러한 표준의 정의는 매우 성공적이었다. 인터넷은 단일 기술로는 역사상 그 어느 기술보다 빠르게 성장하고 있으며 철도나 전등, 전화, 텔레비전 등의 발전 속도를 훨씬 능가하고 있다. 그러나 이는 단지 시작일 뿐이다. 이와 같은 발전은 자발적으로 이루어진 표준과 함께 이루어져 왔는데, IETF의 표준은 자발적으로 수용되는 것이기 때문에 어떠한 나라에서도 이 표준을 수용하지 않을 수 있다. 경쟁적인 여러 개의 표준이 새롭게 나타나고 사라지는 과정을 통해서 IETF의 표준은 더욱 확고하게 성장했다. 물론 IETF의 표준이 모두 성공적이었던 것은 아니다. 현실 세계의 요구에 부합되는 것만이 살아남아서 이름 그대로 진정한 표준이 되었다.

IETF와 그 표준이 성공적일 수 있었던 것은 오픈 소스 공동체가 성공하는 것과 같은 맥락에서 그 이유를 찾을 수 있다. IETF의 표준화 과정은 관심 있는 사람은 누

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

이 글에서는 IETF에 대한 간략한 역사를 소개하고 IETF의 조직과 업무 처리 방법을 검토해 본 뒤에 마지막으로 공개 표준과 공개 문서, 그리고 오픈 소스에 대한 몇 가지 중요한 생각을 덧붙여보기로 한다.

IETF의 역사

IETF는 미국 정부로부터 기금을 지원받던 연구자들의 분기별 모임으로 1986년 1월부터 시작되었다. 같은 해 10월에 있었던 네 번째 모임에는 비정부 업체의 대표들이 초청되었으며, 이때부터 IETF의 모든 모임은 참석을 희망하는 모든 사람에게 공개되었다. 초기 단계에는 그 규모가 크지 않아서 처음 5번의 모임에 평균적으로 35명 정도가 참석했고, 13번의 모임 중에서 가장 참석자가 많았던 1989년 12번째 모임의 경우에도 120명이 왔을 뿐이었다. IETF는 그 후로 크게 성장하여 1992년 3월 23차 모임에 500명이 참석하였고 1994년 3월 29차 모임에는 750명, 1994년 12월 31차 모임에는 1,000명이 참석하면서 1996년 12월 37차 모임에는 참석자가 거의 2,000명에 육박하게 되었다. 이때부터 다시 증가율이 낮아져서 1998년 12월 43차 모임에는 2,100명이 참석했는데 이러한 과정에서 IETF는 1991년부터 모임 회수를 연간 4회에서 3회로 줄이게 되었다.

IETF는 미국 버지니아주 레스톤Reston에 소규모 사무국[02]을 두고 있으며 RFC 편집국[03]은 남가주 대학University of Southern California의 정보 과학 연구소에서 관리한다. IETF는 지금까지 단 한 번도 법인의 형태로 구성되지 않았다. 법적인 실체 없이 단지 활동만 해왔을 뿐이다. 1997년 말까지 IETF의 운영 자금은 미국 연방 정부의 기금과 IETF 회원들의 참가비로 충당되었으며, 1998년부터는 ISOCInternet SOCiety[04]의 지원과 회원들의 참가비로 충당하고 있다.

ISOC는 IETF의 표준화 활동이 원만하게 진행될 수 있도록 법률적인 지원을 제공하고 IETF 관련 사업에 기금을 지원하기 위한 목적으로 1992년에 만들어졌다. ISOC는 회원제로 운영되는 국제적인 비영리 단체며 인터넷이 보급되지 않은 국가에 인터넷을 전파하기 위해서 노력한다. 이제 IETF를 가장 정확하게 정의한다면 ISOC의 지원 아래 표준화를 진행하는 기구라고 말할 수 있다. ISOC의 홈페이지는 http://www.ISOC.org며 IETF의 홈페이지는 http://www.IETF.org다.

워킹 그룹에 대한 개념은 1987년 2월에 있었던 IETF의 15차 모임에서 제안되었는데 현재는 110개가 넘는 워킹 그룹들이 IETF 산하에서 활동한다.

IETF의 구성과 특성

IETF는 회원의 자격에 대한 명시적인 규정이 없는 회원제 단체라고 할 수 있다. 기관이 아닌 개인이나 기업이 회원으로 참여할 수 있다는 정도의 구분 외에는 회원의 자격과 종류에 대한 별다른 기준이 없으며, IETF의 모임이나 메일링 리스트에 참여하는 모든 개인은 IETF의 회원이 될 수 있다.

이 글을 쓰는 현재 시점에서 IETF에는 공식적으로 115개의 워킹 그룹이 존재하며, 워킹 그룹들은 응용, 일반, 인터넷, 운영 및 관리, 라우팅, 보안, 트랜스포트, 사용자 서비스의 8개 분야[05]로 편성되어 있다. 각각의 분야는 에어리어 디렉터Area Director라는 직함의 1명 또는 2명의 자원자가 운영을 맡고 있으며, 에어리어 디렉터들은 IETF의 의장과 함께 IESGInternet Engineering Steering Group라는 이름의 심의 조정단을 구성한다. IESG는 IETF가 제정한 표준을 의결하는 역할을 담당한다. 또한 12명의 인원으로 구성된 인터넷 아키텍처 위원회 IABInternet Architecture Board에서는 워킹 그룹의 조직 형태와 워킹 그룹의 작업 결과에 대한 아키텍처 관련 자문을 제공한다.

IAB의 임원과 에어리어 디렉터의 임기는 2년이며 임명 위원회에 의해서 선출된다.

임명 위원회는 IAB와 에어리어 디렉터를 선출할 시점을 기준으로 이전에 열렸던 3번의 모임에서 최소한 2번 이상 참석한 사람 중에서 자원자를 무작위로 선출하여 매년 새롭게 구성한다.

IETF 워킹 그룹

IETF와 다른 표준화 기구들 사이의 중요한 차이 중 하나는 IETF가 매우 상향적인 구성 방법을 갖고 있다는 점이다. IESG나 IAB가 중요하다고 판단한 현안을 처리하기 위해서 워킹 그룹을 직접 만드는 일은 거의 없다. 대부분의 워킹 그룹은 특정한 문제에 관심 있는 사람들이 소규모 모임을 먼저 만든 다음에 에어리어 디렉터에게 워킹 그룹의 생성을 제안하는 과정을 통해서 이루어진다. 이러한 형태는 IETF가 미래의 작업에 대한 사전 계획을 세울 수 없다는 것을 의미하기도 하지만, 한편으로는 워킹 그룹을 성공적으로 이끌기 위한 회원들의 열정과 전문 지식을 충분히 보장해줄 수 있는 방법이라고 할 수 있다.

새로운 워킹 그룹을 만들기 위해서는 워킹 그룹에 대한 헌장charter을 작성해야 하는데, 이 작업은 워킹 그룹을 만들려고 하는 발기인들과 해당 분야의 에어리어 디렉터 사이의 협의를 통해서 이루어진다. 헌장에는 워킹 그룹에 필요한 조건, 다른 그룹과의 교류 방법, 그리고 수행할 작업의 범위와 한계가 포함되어 있어야 한다. 헌장 작성이 완료되면 IESG와 IAB의 메일링 리스트로 올려져서 검토가 진행된다. 일주일 동안 심각한 이의가 제기되지 않으면, 유사한 분야의 작업이 진행되는지의 여부를 확인하기 위해서 IETF 메일링 리스트와 다른 표준화 기구와의 의사 교환을 위한 메일링 리스트에 올려진다. 그 후 몇 주간의 의견 수렴 기간을 거친 뒤 에 IESG가 제출된 헌장을 승인하면 새로운 워킹 그룹이 만들어지게 된다.

따라서 헌장은 워킹 그룹 자체의 운영 규범이면서 워킹 그룹을 만들기 위한 승인서의 성격도 함께 갖고 있다고 할 수 있다. 헌장은 일반적으로 charter라는 단어로 표현하며 IESG로 제출된 charter(명사로 쓰임)를 검토해서 워킹 그룹의 생성을 승

인해주는 것을 charter(동사로쓰임)한다고 표현한다. IETF에 현재 115개의 워킹 그룹이 있다면 115개의 charter된 워킹 그룹이 있다고 표현할 수 있다. 헌장과 워킹 그룹에 대한 사항은 RFC 1603 문서[06]를 통해 좀 더 자세하게 참고할 수 있다.

IETF 문서

IETF의 모든 문서는 인터넷을 통해서 자유롭게 이용할 수 있도록 공개되어 있다. IETF는 모든 사람이 문서를 자유롭게 이용할 수 있도록 보장하기 위해서 문서가 출판[07]될 때 저자로부터 제한적인 저작권을 양도받는데 이러한 저작권 아래에서는 누구라도 IETF 문서들을 재출판할 수 있으며, IETF의 표준화 과정에서 2차적 문서를 생성할 수도 있게 된다(문서의 원저자는 시일이 경과한 뒤에도 이러한 협의 사항을 철회할 수 없다). 이 외의 다른 권리는 모두 저자가 갖게 된다.

IETF의 활동을 위해서 기본적으로 출판되는 문서들은 RFC다. RFC는 원래 특정한 사안에 대해서 다른 사람들의 견해을 묻는다는 뜻의 'Request For Comments'의 약자로 사용된 말이지만 RFC로 출판될 문서들은 출판되기 이전에 이미 밀도 높은 검토 과정을 마치기 때문에 출판된 이후에도 RFC, 즉 '의견 요청'이라고 하는 것에는 무리가 있다. 따라서 RFC는 마치 고유 명사처럼 RFC라는 이름 그 자체로 받아들이는 것이 가장 바람직할 듯 하다. RFC 문서는 크게 표준 부문과 비표준 부문의 두 가지 영역으로 구분할 수 있는데 표준 부문의 문서들을 이해하기 위해서는 먼저 RFC 문서의 성숙 단계를 이해할 필요가 있다. 표준화를 진행하기 위해서 특정한 표준안을 워킹 그룹에 제안하면, 이 표준안을 제안 표준이라 하고 여기에 관련된 문서들을 '제안 표준 단계 문서'라고 말한다. 제안 표준은 다양한 의견 수렴 과정을 거쳐서 완전한 표준으로 인정되기 전 단계인 준(準) 표준이 되는데 준 표준에 관계된 문서들은 '준 표준 단계 문서'라고 한다. 종국적으로 제안된 표준이 준 표준 상태를 거쳐 완전한 인터넷 표준이 되면 이를 인터넷 표준이라 하고 여기에 관련된 문서들을 '인터넷 표준 문서'라 한다. 따라서 표준 부문의 RFC 문서들은 표준화 과

정의 진행 단계에 따른 구분이라 할 수 있으며 '제안 표준 단계 문서', '준 표준 단계 문서', '인터넷 표준 문서'를 IETF에서는 각각 'Proposed Standard status RFC', 'Draft Standard status RFC', 'Internet Standard RFC'라는 용어로 표현한다. 비표준 부문에는 표준화 진행 과정에 직접 연결되지 않는 그 외의 문서들이 포함되는데 정보를 제공하기 위한 '정보 문서'와 실험이나 검증 결과를 담는 '실험 문서', 더 이상 실제로 적용되지 않고 단지 보관되는 과거의 문서들인 '역사 문서'들이 여기에 포함된다. 비표준 부문의 문서들인 '정보 문서'와 '실험 문서', '역사 문서'는 각각 'Informational status RFC', 'Experimental status RFC', 그리고 'Historic status RFC'라는 용어로 표현된다.

IETF는 RFC 이외에 인터넷 드래프트InterNet-Drafts라는 문서도 사용한다. 이 문서는 RFC의 원래 의미인 'Request For Comments'의 목적으로 사용되는 임시 문서며 6개월 후에 자동으로 폐기된다. 인터넷 드래프트 문서들은 표준화 작업이 진행되는 과정 이외에는 인용되거나 참고되지 않는다.

IETF의 운영 과정

IETF의 모토는 '대략적인 합의와 표준의 실제적인 구현'이라고 할 수 있다. 특정 안건이 채택되기 위해서 워킹 그룹의 만장일치 찬성이 요구되지는 않지만 대부분의 워킹 그룹 회원들에게 그 필요성을 입증하지 못하는 안건은 승인될 수 없다.

안건이 승인되기 위해서 필요하다고 명시된 찬성률은 없다. 그러나 일반적으로 90% 이상의 찬성을 얻으면 승인되고 80% 미만일 경우에는 부결된다고 할 수 있다. IETF 워킹 그룹은 실제로 투표하지는 않지만 거수[08]를 이용해서 찬반을 확인할 수 있다. 대부분의 다른 표준화 기구에서는 만장일치에 가까운 합의에 의해서만 표준이 채택되지만 IETF에서는 다수의 찬성에 의한 대략적인 합의를 통해서 표준이 채택된다.

또한 표준으로 완전히 인정되기 이전에 실제적인 구현이 이루어져야 하는데, 이러한 두 가지 사항을 IETF 안에서는 'rough consensus and running code', 즉 '대략적인 합의와 실제적인 구현'이라 표현한다.

비표준 부문의 RFC 문서들은 IETF 워킹 그룹의 활동을 통해서 만들지만 자신의 생각이나 기술적 제안을 인터넷 공동체에 알리고 싶은 개인이 만들 수도 있다. RFC로 출판되기 위한 거의 모든 문서들은 IESG에서 검토 과정을 거치며 IESG는 RFC 편집국에 출판의 허용 여부에 대한 자문을 제공한다. 이러한 과정을 거쳐 RFC 편집국은 문서의 출판 여부를 결정하게 되는데, 경우에 따라서 IESG로부터 별도의 문서가 첨부된 경우에는 이 문서를 RFC 안에 포함시킬지도 결정하게 된다. IESG가 별도의 문서를 첨부하는 경우는 제안된 사항이 만족할만하지 못했다는 것을 의미하며 여기에 따른 주의를 주는 것이 도움되리라고 판단했을 때다.

표준 부문 문서의 경우에는 RFC로 출판되기 전에 인터넷 드래프트의 형태로 먼저 만들어지는 것이 일반적인 형태다. IETF의 초기 작업은 워킹 그룹을 통해서 이루어지므로 인터넷 드래프트 또한 워킹 그룹에 의해서 생성되며 인터넷 드래프트를 거쳐서 제안 표준 단계가 된다. 워킹 그룹에서 제안 표준을 평가하는 마지막 과정을 라스트 콜last call이라고 부르는 데, 라스트 콜은 워킹 그룹 의장의 주관으로 제안 표준에 문제가 될만한 사항이 있는지 확인하기 위해서 메일링 리스트를 통해 약 2주간의 일정으로 진행된다. 만약 라스트 콜을 통해서 제안 표준이 워킹 그룹 대다수가 찬성하는 대략적인 합의rough consensus로 승인되면 제안 표준은 IESG의 승인을 받기 위해 IESG로 보내진다. IESG 평가의 첫 번째 단계는 라스트 콜을 IETF 전체로 확대하는 것이다. IESG의 라스트 콜은 IETF 전체 메일링 리스트로 보내져서 해당 워킹 그룹에 참여하지 않은 사람들도 여기에 대한 의견을 개진할 수 있도록 한다. 일반적으로 IETF 산하의 워킹 그룹에서 보내진 제안 표준의 라스트 콜에는 2주 정도의 시간이 소요되며 그 밖의 경우에는 4주 정도의 시간이 소요된다.

IESG는 IETF 전체 라스트 콜의 결과를 제안 표준에 대한 심의 기준으로 삼는다. IESG는 제안 표준 문서를 승인하고 문서에 대한 출판을 RFC 편집국에 요청하거나 심의 결과를 토대로 문서를 개정하기 위해 저자에게 반송할 수 있는데, 이러한 과정은 표준화 진행 과정에 따른 제안 표준, 준 표준, 인터넷 표준 문서의 심의 과정에서 모두 동일하게 적용된다.

표준에 대한 제안은 일반적으로 표준 부문 문서에 해당하는 '제안 표준 단계 문서'의 형태로 작성되어 표준화 과정에 들어가지만 기술에 대해 불확실한 점이 있거나 추가적인 확인이 필요하다고 판단될 때는 비표준 부문 문서에 해당하는 실험 문서로 작성될 수도 있다. 제안 표준 단계는 기술적인 결함이 없다는 것을 의미한다. 최소 6개월 간의 제안 표준 단계를 거친 문서는 준 표준 단계로 넘어가게 된다. 준 표준 단계의 문서는 반드시 문서의 내용이 명확하고 저작권과 관련된 사항이 모두 해결되었음을 입증해야 하는데, 이러한 사항은 최소한 2개 이상의 완전히 독립적이고 공유 가능한 실제적인 구현을 통해서 입증되어야 한다. 이때 저작권과 관련된 문제가 있을 경우에는 라이선스 과정을 거쳐 구현이 이루어지도록 한다. 구현 단계에서 유의해야 할 것은 개별적인 구현에 사용된 프로토콜을 다중 구현이 가능하게 만들어야 한다는 점이다. 이러한 요구를 충족시키지 못하는 구현은 표준에서 제외된다. 따라서 IETF의 표준은 세부적으로 많은 구현이 나타날 수 있지만, 이러한 요구 조건을 충족시켜야 표준으로 남아 있을 수 있으므로 표준화 과정을 통해서 좀 더 간결하게 다듬어지게 된다. IETF의 모토에 포함되어 있던 '실제 구현running code'이란 바로 문서 상의 표준안으로 표준화가 진행되지 않고 실제로 구현되는 것을 지칭하는 것이다. 모든 제안 표준은 실제 구현 단계를 거쳐야만 비로소 준 표준 상태로 넘어갈 수 있다.

IETF 표준화 과정의 마지막 단계는 인터넷 표준이다. 최소한 4개월 이상의 준 표준 상태를 거치면서 충분한 시장성을 입증하게 되면 제안된 표준은 마침내 인터넷 표준으로 간주한다.

다른 표준화 기구의 표준화 진행 과정과 비교해볼 때 IETF에는 중요한 두 가지 차이점이 있다. 첫째는 최종적으로 결정된 표준의 주요 부분 대부분이 제안 표준 상태와 거의 동일하다는 점이다. 왜냐하면 제안 표준은 실제적으로 구현만 되지 않았지 드래프트 상태부터 많은 검증을 거쳐서 이미 문제가 없다고 판단된 유용한 기술이기 때문이다. IETF에서는 제안 표준이 준 표준으로 넘어가는 단계에서 실제 구현이 요구될 뿐이다. 둘째는 대략적인 합의다. 만장일치보다 대략적인 합의를 거치면 문제를 일으킬 소지가 없는 좀 더 깔끔한 제안이 될 수 있다.

정리하면 IETF는 상향식 처리 방식과 "구입하기 전에 테스트해본다."[09]는 두 가지 원칙에 의해서 운영된다고 할 수 있다. RFC 문서를 통해 인터넷 표준을 만들어 가는 과정은 RFC 1602 문서에 좀 더 자세하게 설명되어 있다.

공개된 표준과 문서, 그리고 오픈 소스

IETF의 표준이 지금까지 성공적이었던 주된 요인 중 하나는 IETF의 문서와 표준화 과정에 대한 공개 정책이라고 할 수 있다. IETF는 메일링 리스트와 모임뿐만 아니라 문서 자료를 모두 공개하는 매우 드문 표준화 기구의 하나다. 기존의 표준화 기구 대부분과 심지어 인터넷과 관련된 신생 단체도 문서 자료와 모임을 회원에게만 공개하거나 별도의 비용을 지불한 사람에게만 허용하는 경우가 많다. 때로는 이러한 방법이 표준화 기구를 운영하기 위한 기금을 마련하기 위한 목적으로 사용되기도 하지만 기구 자체가 회비 납부를 기본으로 하는 회원제로 운영되기 때문에 자료에 접근할 수 없는 경우도 있다. 사람들이 이러한 기구에 가입하려는 이유의 하나는 표준화 과정에 참여하거나 그들이 진행하고 있는 표준에 대한 접근을 원하기 때문이다.

표준화 과정에 대한 참여를 제한하는 것은 그렇지 않은 경우에 비해 사용자와 기업의 필요를 충족시키지 못하는 표준을 만들게 될 가능성이 커지고, 운영자들이 표준을 합리적으로 지원하지 못할 정도로 복잡해질 가능성 또한 커진다고 할 수 있다.

표준화가 진행되는 문서에 대한 접근을 제한하는 것은 표준을 구현할 사람들이 특정 부분에 대한 세부 사항과 내역을 이해할 수 없게 만들기 때문에 결과적으로 표준에 결함이 발생할 수 있다. 또한 완성된 표준에 대한 접근을 제한하면 학생과 개발자들이 표준을 이용하고 활용할 때 미미한 출발점에 설 수밖에 없게 된다.

IETF는 오픈 소스 운동이 형성되기 오래전부터 이미 오픈 소스의 개념을 지원해 왔다. 최근까지도 표준 문서 단계의 진행을 위해 필요한 다중 구현 방식의 일부로 IETF의 기술에 대한 참조 구현[10]이 이루어지는 것은 일반적인 일이었다. 이러한 방법이 IETF의 표준화 과정에서 공식적으로 필요한 것은 아니지만 이러한 과정을 통해서 매우 유용한 부산물(副産物)들을 얻을 수 있다. 안타깝게도 참조 구현의 방법은 표준이 점점 더 복잡해지고 구현에 소요되는 비용이 늘어남에 따라 줄어드는 것이 현실이다. 참조 구현을 사용하는 이러한 관행이 IETF 안에서 없어진 것은 아니지만 오픈 소스 운동이 이 방법을 활성화한다면 매우 유익할 것이다.

현재까지는 가시적으로 명확하게 드러나지 않지만 공개 문서와 공개된 표준화 과정은 오픈 소스 운동에 있어 필요 불가결한 부분이다. 표준 문서가 워킹 그룹을 통해 연결되는 방법처럼 현재 진행되는 일에 대한 명확한 동의가 없다면, 오픈 소스 운동처럼 프로젝트의 진행이 분산적으로 이루어지는 경우에는 혼란을 겪거나 실패하기 마련일 것이다. 공개 문서와 공개된 표준화 과정, 그리고 오픈 소스 사이에는 본질적인 연관성과 협력 관계가 있다. 이러한 협력 관계가 지금의 인터넷을 만들었으며 또한 미래의 또 다른 경이로운 것들을 만들어 낼 것이다.

---

01 역자주_태스크포스란 특정한 작업을 수행하기 위해 구성된 '전담 대응팀'을 말한다.

02 역자주_RFC 사무국에 대한 정보는 http://www.ietf.org/secretariat.html 참고.

03 역자주_'RFC 편집국'으로 번역한 RFC 에디터(RFC Editor)의 역할은 RFC 문서를 최종 점검하고 이를 인터넷상에서 공개하는 것이다. 또한 인터넷에서 RFC 문서를 검색할 수 있도록 RFC 색인 파일을 관리하는 일도 맡고 있다. 1999년 10월 16일에 RFC 편집국의 책임을 맡았던 조너선 포스텔(Jonathan B.Postel) 박사가 향년 55세의 나이로 사망했는데, IANA(Internet Assigned Numbers Authority)와 ISOC(Internet SOCiety)의 활동에 핵심적인 역할을 했던 그의 추모 홈페이지는 http://www.postel.org다. RFC 편집국의 홈페이지는 http://www.rfc-editor.org다.

04 역자주_Internet Society는 우리말로 인터넷 협회 또는 인터넷 협의회쯤으로 번역할 수 있겠지만약어인 ISOC가 널리 통용되고 있으므로 별다른 번역 없이 ISOC라고 표기한다.

05 역자주_IETF 홈페이지를 통해서 워킹 그룹을 검색할 경우에는 http://www.ietf.org/html.charters/wg-dir.html을 참고할 수 있으며, 다음과 같은 8개의 영역으로 워킹 그룹을 구분하고 있다. Applications(응용), General(일반), Internet(인터넷), Operations and Management(운영 및 관리), Routing(라우팅), Security(보안), Transport(트랜스포트), User Service(사용자 서비스)가 그것이다.

06 역자주_RFC 문서를 검색할 수 있는 추천할만한 사이트로는 http://www.faqs.org/rfcs가 있다.

07 역자주_ 출판이라는 용어는 일반적으로 서적을 인쇄해서 배포하는 것을 의미하지만, 단지 서적의 형태뿐만 아니라 온라인이나 CD-ROM 등과 같은 전자 매체를 통한 배포도 출판의 범위에 포함된다.

08 역자주_실제 회의를 통한 참석자들의 거수를 의미하는 것은 아니다. 메일링 리스트를 통해서 투표가 직접 진행되지는 않지만 찬성이나 반대 의사를 개진하는 인원수에 따라 대략적인 지지율을 산출할 수 있다.

09 역자주_"구입하기 전에 테스트해본다."는 뜻의 'flybeforeyou buy'라는 표현은 몇 년 전부터 미군이 도입한 전략 무기 구입 정책의 변화에서 유래한 말이다. 무기 구매 계획은 대부분 실제 테스트 없이 서류 검토만으로 이루어졌지만 현재는 대량의 항공기를 구매하기 전에 반드시 사전테스트를 거쳐서 기종을 선정하고 있다. 이러한 'flybefore you buy' 정책은 표준화 과정에서 실제적인 구현이 이루어지도록 하는 IETF의 'running code' 원칙과 일맥상통한다.

10 역자주_참조 구현(reference implementation)이란 제안된 표준을 시험해볼 수 있게 하려고 완성된 형태의 실체로 구현하는 것이다. 예를 들면 CGI 인터페이스에 대한 새로운 표준이 제안되었을 때, 다양한 종류의 구현 방법이 제시되어 표준화 과정에서 이들을 포괄할 수 있는 다중 구현(multiple implementation)으로 통합되는데, 아파치 서버와 리눅스와 같은 특정한 플랫폼에서 구동될 수 있도록 제안된 표준을 이용해서 만들어진 하나의 CGI 검색 프로그램은 다중 구현에서 인정될 수 있는 참조 구현이 된다.

4 | GNU 운영체제와 자유 소프트웨어 운동

리처드 스톨만Richard Stallman

송창훈 역

소프트웨어를 공유했던 최초의 공동체

MIT 대학의 인공지능 연구소에서 일하기 시작했던 1971년부터 나는 소프트웨어를 공유하는 공동체의 일원이 되기 시작했다. 이 공동체는 이미 수년 전부터 존재하던 것이었으며 MIT에만 유일하게 존재하는 특별한 것은 아니었다. 요리법을 공유하는 것이 요리의 역사 만큼 오래된 것처럼 소프트웨어를 공유하는 것은 컴퓨터의 역사만큼이나 오래된 것이었다. 그러나 우리는 소프트웨어를 공유하는 더욱 이상적인 우리만의 공동체를 MIT에서 만들어 나갔다.

인공지능 연구소에서는 ITSIncompatible Time-sharing System라고 불리는 시분할time-sharing 운영체제를 사용했었는데, ITS는 당시에 사용되던 가장 큰 시스템의 하나인 DEC사의 PDP-10 기종에 탑재할 목적으로 인공지능 연구소의 해커 연구원들에 의해서 어셈블리 언어로 만들어진 운영체제였다. 인공지능 연구소의 연구원으로 그리고 공동체를 구성하고 있는 해커[01]의 일원으로 나는 ITS를 더 나은 시스템으로 만드는 일에 종사했다.

우리는 우리들의 소프트웨어를 '자유 소프트웨어free software'라고 말하지는 않았다. 왜냐하면 그 당시에는 '자유 소프트웨어'라는 용어가 아직 존재하지 않았기 때문이다. 하지만 우리가 만들었던 소프트웨어의 모습은 지금의 '자유 소프트웨어'가 의미하는 바로 그것이었다. 다른 대학이나 기업에서 우리가 만든 프로그램을 사용하

거나 다른 시스템으로 이식하기 위해서 우리에게 도움을 요청할 때면 우리는 언제든지 우리의 프로그램을 그들에게 빌려주었다. 이러한 형태는 매우 일반적인 것이어서 새롭고 흥미로운 프로그램을 사용하는 사람을 발견했다면 언제든지 그 사람에게 프로그램의 소스 코드를 보여 달라고 요청할 수 있었다. 공동체 시절 당시는 특정한 프로그램의 소스 코드를 자유롭게 얻을 수 있었기 때문에 프로그램을 수정하거나 해당 프로그램을 기반으로 한 새로운 프로그램을 만들 수 있는 가능성이 언제든지 열려 있었던 공유의 정신이 충만한 시절이었다.

공동체의 붕괴

이러한 상황은 1980년대 초에 DEC사가 PDP-10 시리즈의 생산을 중단하면서부터 급격하게 붕괴되었다. 1960년대를 풍미했던 PDP-10 시리즈의 강력하고 우아한 설계 구조도 1980년대로 접어들면서 주소 공간을 더 이상 자연스럽게 확장할 수 없었기 때문에 DEC사는 32비트 기반의 VAX를 주력 기종으로 삼으면서 PDP-10 시리즈를 퇴역시켰다. PDP-10 시리즈의 단종은 ITS 환경에서 만들어진 거의 모든 프로그램이 호환성의 문제로 인해서 더 이상 사용될 수 없음을 의미했다.

인공지능 연구소를 중심으로 한 해커들의 공동체도 PDP-10과 함께 붕괴되기 시작해서, 1981년에는 인공지능 연구소에서 근무하던 대부분의 해커가 스핀오프[02] 회사인 심볼릭스Symbolics로 직장을 옮기게 되었다. 스티브 레비Steve Levy의 저서인 『해커들』[03]에는 이러한 상황이 전성기 때의 공동체에 대한 일화와 함께 자세히 묘사되어 있다.

해커들의 이직에 따른 가장 큰 문제는 바로 공동체 자체를 유지할 인적 자원이 없어진다는 것이었다. 인적 자원의 부족은 결국 1982년에 인공지능 연구소에서 구입한 PDP-10을 운영하기 위해서 ITS가 아닌 DEC의 운영 체제를 도입하는 데까지 이르렀다. 물론 DEC의 운영체제는 자유 운영체제가 아니었다.

당시에 소개되었던 VAX나 68020과 같은 현대적인 모델의 컴퓨터는 전용 운영체제를 탑재했는데, 이들은 모두 자유 소프트웨어가 아니었다. 따라서 바이너리 형태의 프로그램을 사용하고자 할 때 조차도 관련 자료를 유출하지 않겠다는 비공개 계약 조건에 동의해야만 했다.

이러한 사실은 컴퓨터를 사용하는 처음 단계부터 자신의 주위 사람을 돕지 않겠다고 약속하는 것과 같은 의미를 갖는다. 즉, 상호 협력적인 공동체의 형성이 처음부터 불가능하게 되는 것이다. 독점 소프트웨어의 소유자들은 소프트웨어를 공유하는 것을 '저작권 침해pirate'라는 단어를 사용해서 마치 해적 행위처럼 해악한 것과 동일시하려고 했으며, 프로그램에 대한 수정이 필요할 때는 그들에게 이를 요청하도록 했다.

독점 소프트웨어 제도는 소프트웨어를 다른 사람과 함께 공유하거나 교환하는 것을 금지하기 때문에 결코 사회적인 제도라고 할 수 없다. 이는 분명하게 잘못된 비윤리적인 제도이지만 이러한 나의 주장이 어떤 사람에게는 충격적인 사실로 받아들여질 수도 있을 것이다. 그러나 사용자를 고립시키고 서로를 돕는 것을 금지하는 방법으로 사람을 무력하게 만드는 제도에 대해서 우리가 언급할 수 있는 것은 과연 무엇이겠는가? 만약 독점 소프트웨어 제도에 대한 나의 생각에 거부감이 느껴진다면 아마도 그 사람은 독점 소프트웨어 산업이 주입했던 개념을 아무런 의심 없이 그대로 인정해왔기 때문이라고 할 수 있다. 소프트웨어 판매자들은 소프트웨어의 유통 형태에 대해서 기존의 방식이 유일한 방법이라는 생각을 사람들에게 각인하기 위해서 오랫동안 노력해왔다.

소프트웨어 제작업자들이 사용하는 "저작권 침해 행위를 중단하라."나 "우리의 권리를 행사한다."는 등의 표현은 표면일 뿐이며, 그들이 의도하는 숨겨진 진의는 법률적이고 과격한 표현을 사용함으로써 대중들로 하여금 그들의 주장을 비판 없이 수용하게 만드려는 데 있다.

구체적인 예를 살펴보면 일반 사용자가 소프트웨어 회사의 권리를 '침해'한다든가 또는 소프트웨어 회사들이 그들의 정당한 '권리를 행사'한다는 표현 안에는 마치 소프트웨어 제조 회사가 소프트웨어를 유형, 무형으로 완전히 소유할 수 있는 천부적인 자연권을 갖고 있어서 소프트웨어에 대한 사용자의 권리를 통제할 수 있다는 전제가 포함되어 있다.

만약 그들의 권리가 진정한 자연권이라면 대중이 그들로부터 어떠한 피해를 입더라도 이에 반대하거나 저항할 수 없을 것이다. 그러나 한 가지 흥미로운 사실은 미국의 헌법과 법률 전통 안에서는 저작권이 자연권 안에 포함되지 않음에도 불구하고 연방 정부가 인위적으로 부여한 독점권에 의해서 소프트웨어를 자유롭게 복제할 수 있는 사용자들의 권리가 제한당하고 있다는 점이다.

소프트웨어 제조 회사들이 사용하는 표현에 내포되어 있는 또 하나의 가정은 소프트웨어에 있어서 중요한 것은 오직 소프트웨어를 이용해서 수행할 수 있도록 사용자들에게 허용된 작업 자체이지 작업 과정을 통해 서 형성될 수 있는 사람들간의 만남과 교류는 아니라는 점이다. 만약 이러한 점들이 중요하게 고려되었다면 사람들의 만남과 교류를 통해서 소프트웨어가 공유될 수 있다는 무척이나 자연스러운 발상에 근거해서 소프트웨어를 공유하거나 소스 코드를 보여주는 행위를 '권리의 침해'라는 강압적인 표현으로 오도하지는 못할 것이다. 그들이 중요하게 생각하는 것은 소프트웨어를 이용해서 최대한의 재화를 창출하는 것이므로 소프트웨어에 대한 적용 범위와 고려의 대상을 사용자의 만남과 교류가 아닌 작업 그 자체로 한정시킨 것이다.

또한 '권리의 침해'나 '권리의 주장'이라는 표현 안에는 소프트웨어를 창작하고 수정하는 등의 모든 권한이 소프트웨어 제조 회사에 귀속되어 있으므로 그들의 주장을 인정하고 수용하지 않으면 특정한 작업을 수행할 수 있는 프로그램이나 그 밖의 유용한 소프트웨어를 전혀 사용할 수 없게 되리라는 강압적인 또 하나의 잘못된 가

정이 포함되어 있기도 하다. 이러한 가정은 매우 설득력 있게 들릴 수도 있지만 자유 소프트웨어 운동이 구속과 통제가 없는 많은 양의 자유 소프트웨어를 만들면서 그 오류를 불식할 수 있었다.

만약 이러한 가정을 거부하고 사용자를 최우선으로 고려한 일반적인 상식과 윤리의 기준에서 이 문제를 판단한다면 우리는 매우 다른 결론에 도달하게 된다. 사람이 서로 돕는 것은 이 사회를 지탱해 나가는 근간이므로 결국 사용자는 그들의 필요에 따라서 프로그램을 자유롭게 수정하거나 공유할 수 있는 것이다.

이러한 결론이 도출되는 좀 더 엄밀한 추론 과정은 지면 관계상 생략하기로 한다. 그러나 GNU 홈페이지에 올려져 있는 http://www.gnu.org/philosophy/why-Free.ko.html를 통해서 좀 더 자세한 내용을 참고할 수 있다.

냉혹한 도덕적 선택

공동체가 사라짐에 따라서 이 공동체를 예전처럼 지탱하는 것은 불가능했고, 나는 대신 냉혹한 도덕적 선택의 기로에 서게 되었다. 손쉬운 선택은 비공개 협약에 서명하고 동료 해커들을 돕지 않음을 약속하면서 독점 소프트웨어의 세계에 합류하는 것이었다. 만약 그렇게 했다면 내가 개발했을 소프트웨어 또한 비공개 협약에 의해서 배포되었을 것이므로 다른 사람이 동료를 돕지 못하도록 강요하는 또 하나의 요인이 되었을 것이다.

나는 이러한 방법으로 돈을 벌 수 있었을 것이며, 아마도 프로그램을 만드는 작업으로부터 즐거움을 누릴 수도 있었을 것이다. 그러나 훗날 내 인생을 정리하면서 내가 보낸 과거를 돌이켜 봤을 때 내가 사람들을 단절시키는 벽을 만들었으며, 또한 이 세상을 더 나쁘게 만들어왔다는 것을 느끼게 될 것이라는 점도 알고 있었다.

나는 MIT의 인공지능 연구소와 나에게 제어 프로그램의 소스 코드를 줄 것을 거부했던 사람으로 인해서 프린터의 사용에 곤란을 겪은 경험을 갖고 있었기 때문에,

이러한 비공개 협약이 가져올 결과를 이미 알고 있었다(프로그램에 특정한 기능이 결여되어 있었기 때문에 프린터를 사용하는 데 많은 곤란을 겪었다). 그래서 나는 비공개 협약은 결코 유익하지 않다는 생각을 갖고 있었다. 그가 소스 코드를 함께 공유하기를 거부했을 때 나는 매우 분개했으며 나는 결코 같은 일을 다른 사람에게 하지 않겠다고 다짐했던 것이다.

좀 더 간단하지만 결코 유쾌하지 않은 또 하나의 선택은 컴퓨터 분야를 떠나는 것이었다. 이 선택은 내가 가진 기술이 내가 원하지 않는 방향으로 사용되지 않도록 할 수 있지만 다른 측면에서 보면 내가 가진 기술이 허비되는 것이라고 할 수 있다. 만약 이 방법을 선택했다면 컴퓨터 사용자들을 제한하고 단절시키는 일을 직접 하지는 않았을 것이므로 내가 비난의 대상이 되지는 않았겠지만, 그럼에도 불구하고 그러한 일들은 계속해서 일어났을 것이다.

이러한 이유로 인해서 나는 프로그래머로서 뭔가 좋은 역할을 할 수 있는 방법을 찾게 되었다. 그러면서 공동체를 다시 부활시키기 위해서 내가 직접 만들 수 있는 프로그램이 없을까라는 생각을 하기에 이르렀다.

해답은 명확했다. 제일 먼저 필요한 프로그램은 바로 운영체제였다. 운영체제는 컴퓨터를 사용하기 위해서 필요한 가장 핵심적인 소프트웨어다. 운영체제가 있다면 컴퓨터를 이용해서 다양한 종류의 작업을 할 수 있지만, 운영체제가 없다면 컴퓨터 자체를 사용할 수 없게 된다. 따라서 자유롭게 사용할 수 있는 운영체제를 가질 수 있다면 상호 협력적인 해커들의 공동체를 다시 한번 재건할 수 있을 것이며, 누구에게나 공동체에 합류하기를 권유할 수 있게 될 것이다. 또한 운영체제의 구입에 수반되는 '재배포를 금지한다'는 등의 계약 조건에 구애받을 필요가 없어지므로 친구의 자유를 침해하지 않고도 누구나 컴퓨터를 사용할 수 있게 될 것이었다.

운영체제의 개발자로서 나는 이러한 작업에 적합한 기술을 갖고 있었다. 그래서 성공하지 못할 가능성이 있음에도 불구하고, 나는 이 일이 내 소명이라고 생각하

게 되었다. 나는 새롭게 개발할 운영체제를 유닉스 와 호환되도록 만들 생각을 했다. 그렇게 함으로써 이식성을 갖게 되고 기존의 유닉스 사용자들이 새로운 시스템에 쉽게 적응할 수 있으리라고 생각했기 때문이다. GNU라는 이름은 MIT의 해커들이 프로그램의 이름을 지을 때 사용하던 재귀적 약어recursive acronym의 습관을 반영한 것으로 GNU's Not Unix, 즉 "GNU는 유닉스가 아니다."라는 뜻이 되도록 GNU라는 단어를 처음부터 의도적으로 조합해서 만든 것이다. 예를 들면, EINE라는 이름의 에디터는 Emacs가 아니라는 의미로 "Eine Is Not EMacs."라는 문구를 먼저 생각한 뒤에 여기서 각 단어의 첫 글자를 따서 EINE라는 이름이 되도록 만든 것이며, CYGNUS는 'Cygnus Your GNU Support'라는 문구를 이용해서 만든 것이다. 따라서 GNU는 유닉스와 호환되도록 만들어진 운영체제지만 유닉스와는 다른 운영체제라는 의미를 내포시키기 위해서 만든 이름이라고 할 수 있다.

운영체제는 단순히 커널만을 의미하는 것이 아니라 여러 가지 프로그램을 실행시킬 수 있는 통합적인 시스템 운영 환경을 의미하는 것이다. 1970년대에 운영체제라고 말할 만한 것들은 모두 셸과 같은 명령어 처리기와 어셈블러, 컴파일러, 인터프리터, 디버거, 텍스트 에디터 그 고 메일 프로그램과 같은 다양한 종류의 프로그램들을 갖고 있었다. ITS와 멀틱스Multics, VMS, 유닉스 등의 운영체제도 이러한 프로그램을 포함했고 따라서 이제부터 시작될 GNU 운영체제에도 이러한 프로그램이 모두 필요했다.

훗날 나는 힐렐Hillel [04]이 말했다는 다음과 같은 잠언을 듣게 되었는데, GNU 프로젝트를 시작하게 된 결단은 이러한 정신과 일맥 상통한다고 볼 수 있다.

> 내가 내 자신을 위하지 못한다면 그 누가 나를 위해 줄 것인가?
>
> 내가 내 자신만을 위한다면 온전한 나 자신이 될 수 있을까?
>
> 바로 지금이 아니라면 그 언제 할 수 있을 것인가?

구속되지 않는다는 관점에서의 자유

자유 소프트웨어라는 용어는 때로 잘못 이해되기도 하는데, 이 말은 무료나 공짜라는 말이 내포하는 금전적인 측면과는 전혀 관계 없는 '구속되지 않는다'는 관점에서의 자유를 의미한다. 다른 언어와는 달리 영어에는 무료gratis를 의미하는 단어와 자유Freedom를 의미하는 단어가 별도로 존재하지 않고 모두 Free라는 단어를 사용하기 때문에 이러한 오해의 소지가 더욱 많다고 할 수 있다. 무료 맥주free beer나 언론의 자유free speech와 같은 예를 생각해보면 그 차이를 보다 명확하게 구분할 수 있을 것이다.[05] 따라서 사용자에게 다음과 같은 네 가지 종류의 자유를 실질적으로 보장하는 프로그램을 자유 소프트웨어라고 말할 수 있다.

목적에 상관 없이 프로그램을 실행시킬 수 있는 자유

필요에 따라서 프로그램을 개작할 수 있는 자유(이러한 자유가 실제로 보장되기 위해서는 소스 코드를 이용할 수 있어야만 한다. 왜냐하면 소스 코드 없이 프로그램을 개작한다는 것은 매우 어려운 일이기 때문이다)

무료 또는 유료로 프로그램을 재배포할 수 있는 자유

개작된 프로그램의 이익을 공동체 전체가 얻을 수 있도록 이를 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 중요하며, 자유 소프트웨어들을 모아서 CD-ROM의 형태로 판매하는 것은 자유 소프트웨어의 발전 기금을 조성할 수 있는 실질적인 방법이 되기 때문에 공동체에 있어서 매우 중요한 부분이다. 따라서 임의의 배포판이나 소프트웨어 모음집에 자유롭게 포함시킬 수 없는 프로그램은 자유 소프트웨어가 아니라고 할 수 있다.

자유라는 단어가 가진 모호함 때문에 사람들은 오랫동안 이를 대체할 수 있는 용어를 생각해왔지만 적절한 용어를 찾지는 못했다.

영어는 다른 언어에 비해서 많은 단어와 뉘앙스를 갖고 있지만 간단명료함을 갖고 있지는 못하다. 자유 소프트웨어라는 용어에서 사용되는 Free라는 단어는 unfettered, 즉 해방(解放)시킨다는 의미와 같은 뜻이라고 할 수 있을 것이다. liberated나 Freedom 또는 Open이라는 단어는 이 단어가 가진 다른 의미나 단점 때문에 Free 대신 사용하기에 문제가 있다고 생각한다.[06]

GNU 소프트웨어와 GNU 시스템

완전한 운영체제 전체를 만든다는 것은 매우 커다란 프로젝트에 해당한다. 따라서 나는 기존에 존재하던 사용 가능한 자유 소프트웨어를 개작하거나 취합해서 시스템을 완성시켜 나갈 생각을 하게 되었다.

예를 들어 프로젝트가 시작되었던 초반에는 텍TeX을 주된 조판 프로그램으로 사용했으며 몇 년 뒤에는 GNU에 맞는 새로운 윈도우 시스템을 개발하는 것 대신에 X 윈도우 시스템을 사용하기로 결정했다.

이러한 결정으로 인해서 GNU 시스템은 단순히 GNU 소프트웨어를 모두 모아 놓은 것 이상이 되었다. GNU 시스템은 GNU 소프트웨어 뿐만 아니라 그들만의 목적을 가진 다른 사람이나 프로젝트를 통해서 만들어진 소프트웨어도 포함하게 되었는데, 이는 이러한 프로그램이 자유 소프트웨어이므로 가능한 것이었다. 일반적으로 '그누'라고 발음하는 GNU라는 이름은 GNU 프로젝트와 GNU 시스템을 지칭하는 데 모두 사용한다. GNU 시스템은 GNU 프로젝트를 통해서 구현하려고 하는 유닉스와 호환되는 완벽한 소프트웨어 시스템 전체를 말하고 GNU 프로젝트는 이러한 시스템을 구현하기 위해서 진행되는 프로젝트 자체를 의미한다.

프로젝트의 시작

1984년 1월 나는 MIT의 연구원직을 사임하고 GNU 소프트웨어를 만들기 시작했다. 내가 MIT를 떠난 이유는 연구원이 개발한 소프트웨어에 대한 저작권을 소속 회사나 학교로 귀속시킬 수 있는 법률과 내규에 따라서 MIT 당국이 내가 개발한 소프트웨어를 자유 소프트웨어로 만들지 못하게 할 가능성이 있었기 때문이다. 만약 내가 MIT의 연구원으로 남아 있으면 그들이 결과물에 대한 소유권을 주장하거나 MIT의 배포 조항을 강제로 적용할 가능성도 있었으며, 심지어 독점 소프트웨어 패키지로 만들 가능성도 배제할 수 없었다. 그러나 나는 공동체 모두가 공유할 수 있는 새로운 소프트웨어를 만들려는 의도가 이러한 외적 조건으로 인해서 좌절되는 것을 원하지 않았다.

한편 내가 연구원직을 사임했음에도 불구하고 MIT의 인공지능 연구소장이었던 패트릭 윈스턴Patrick Winston 교수는 내가 연구실의 장비를 계속해서 사용할 수 있도록 배려해 주었다.

첫 번째 단계

GNU 프로젝트를 시작하기 얼마 전에 나는 VUCK라는 이름으로도 알려진 자유 대학 컴파일러 킷Free University Compiler Kit에 대한 이야기를 듣게 되었다(약어로 사용된 VUCK가 Free의 첫자인 F로 시작되지 않는 것은 Free에 해당하는 독일어가 V로시작되기 때문이다). 이 컴파일러는 C와 파스칼을 포함한 여러 개의 언어를 컴파일할 수 있었고 다양한 종류의 플랫폼에서 사용할 수 있도록 만들어진 것이었다. 나는 VUCK의 개발자에게 GNU가 이 컴파일러를 사용할 수 있는지의 여부를 서신으로 물어보았다.

자유 대학Free University이라는 이름 안에서의 Free는 종교로부터 자유롭다는 의미로 사용된 것이다. 그는 대학은 자유롭지만 컴파일러는 그렇지 않다며 비웃는 듯한

회신을 보내왔다. 즉, 대학의 이름에는 자유라는 단어가 포함되어 있지만 VUCK[07]는 학생들에게 결코 자유로운 소프트웨어가 아니었던 것이다. 이러한 이유로 인해서 나는 GNU 프로젝트를 위한 첫 번째 프로그램으로 다수 언어와 플랫폼을 지원하는 컴파일러를 만들 결심을 하게 되었다.

컴파일러 전체를 모두 작성해야 하는 수고를 피할 수 있으리라는 희망을 갖고 나는 로렌스 리버모어 연구소Lawrence Livermore Lab.에서 개발한 파스텔Pastel 컴파일러의 소스 코드를 입수했다. 이 컴파일러는 멀티 플랫폼을 지원하는 것이었으며, 시스템 프로그래밍 언어로 사용할 목적으로 파스칼 언어를 확장한 파스텔 언어로 만들어진 것이었다. 물론 파스텔 컴파일러는 파스텔 언어를 컴파일하는 데 사용되는 것이다. 때문에 나는 C 언어를 처리하기 위한 프런트 엔드front-end 부분을 덧붙여서 모토로라의 68000 시스템에 맞게 이식하기 시작했다. 그러나 파스텔 컴파일러를 사용하기 위해서 수 MB 용량의 스택stack이 필요하다는 사실을 알게 되었을 때, 나는 파스텔 컴파일러를 활용하려던 계획을 포기할 수밖에 없었다. 왜냐하면 68000 기반의 유닉스 시스템에서는 기껏해야 64KB의 스택만 사용할 수 있었기 때문이다.

포트란을 비롯한 프로그래밍 언어 대부분은 프로그램에서 사용할 변수를 미리 선언하도록 설계되어 있다. 그러나 파스텔 언어는 변수를 사용에 관계 없이 임의의 위치에서 선언할 수 있게 되어 있었으므로 선언한 변수와 사용하는 변수에 대한 정보를 올바르게 유지하려면 프로그램을 한 번에 읽어서 메모리로 모두 올린 뒤에 처리하는 구조를 가질 수밖에 없었고, 이에 따라서 소스 코드와 거의 동일한 크기의 메모리와 스택이 필요한 상황이 일어날 수밖에 없었다. 즉, 파스텔 컴파일러는 입력 파일 전체를 한번에 파싱해서 이를 하나의 신택스 트리로 구성한 다음, 연속된 명령어로 변환해서 출력 파일을 생성하는 구조를 갖고 있었던 것이다. 그러나 이러한 방식에서는 레지스터와 스택을 전혀 효율적으로 사용할 수 없기 때문에 나는 새로운 컴파일러를 처음부터 다시 만들어야 한다는 결론을 내리게 되었다. 파스텔 컴파일러를 포기한 이후에 새로 개발한 컴파일러는 이제 GCC라는 이름으로 알려지

게 되었는데, GCC 컴파일러에는 파스텔 컴파일러의 소스 코드를 전혀 사용하지 않았지만 파스텔에 추가시켰던 C 언어 프런트 엔드는 GCC에도 계속해서 사용하려고 노력했다. 그러나 이러한 일은 모두 몇 년 뒤에 이루어진 것이었고, 그 당시에는 최우선적으로 GNU Emacs를 만드는 일에 집중했다.

GNU Emacs

나는 1984년 9월부터 GNU Emacs 를 만들기 시작했으며 1985년 초에는 제법 쓸만한 상태가 되었다. 나는 vi나 ed의 사용법을 배우는 데 관심이 없었기 때문에 유닉스가 아닌 다른 기종에서 편집 작업을 해왔지만, Emacs를 만든 후에는 유닉스와 다른 기종의 컴퓨터에서 모두 편집 작업을 할 수 있게 되었다.

사람들이 GNU Emacs 를 사용하고 싶어하는 시점이 되었을 때 이 에디터를 어떠한 방법으로 배포할 것인가에 대한 의문이 제기되었다. 물론 내가 사용하던 MIT 컴퓨터의 익명 FTP 사이트에는 올려둔 상태였다(이 컴퓨터의 이름은 prep.ai.MIT.edu였으며 GNU의 주된 FTP 사이트가 되었다. 몇년 뒤에 이 시스템을 퇴역시켰을 때도 우리는 그 이름을 새로운 시스템에서 계속해서 이어나갔다). 하지만 그당시에는 Emacs에 관심을 가진 사람 중에서 인터넷을 통해 FTP를 이용할 수 있는 사람이 거의 없었다. 따라서 어떠한 방법으로 Emacs를 구할 수 있는지를 그들에게 알려 주는 것이 문제의 핵심이었다.

나는 인터넷을 통해서 Emacs를 구해줄 수 있는 친구를 찾아보거나 반송용 우표가 붙어있는 봉투를 테이프와 함께 보내면 PDP-10용 Emacs를 복사해 주겠다고 말했다. 즉, Emacs를 배포하는 데 있어서 금전적인 비용이 우리에게 청구되지 않는 방법을 생각해냈던 것이다. 그러나 당시의 나는 직장을 갖고 있지 못했기 때문에 자유 소프트웨어를 통해서 수입을 얻을 수 있는 방법을 구상하고 있었다. 나는 150달러의 비용을 지불하면 누구에게나 Emacs가 들어 있는 테이프를 우송해 주는 방법을 생각해냈는데, 이러한 판매 방식을 통해서 나는 자유 소프트웨어를 이용

한 사업을 시작하게 되었고, 리눅스에 기반한 GNU 시스템 전체를 담은 배포판을 판매하는 현재의 리눅스 배포판 업체들의 시초가 되었던 것이다.

프로그램은 누구에게나 자유로운가?

프로그램의 원저작자가 자신의 프로그램을 자유 소프트웨어로 공개했더라도 그 프로그램이 프로그램을 입수한 모든 사람에게 자유 소프트웨어로 남아 있게 된다고 할 수는 없다. 예를 들어 저작권이 없는 공개 소프트웨어public domain software는 일종의 자유 소프트웨어라고 할 수 있다. 그러나 공개 소프트웨어는 누구든지 이 프로그램을 개작해서 독점 소프트웨어로 변형시킬 수 있으며, 저작권이 수반된 많은 종류의 자유 프로그램도 수정 버전을 독점 소프트웨어로 만드는 것을 쉽게 허용하는 라이선스에 의해서 배포된다.

이러한 문제의 전형적인 예는 X 윈도우 시스템에서 찾아볼 수 있다. MIT에 의해서 개발된 X 윈도우 시스템은 자유 소프트웨어로 배포되었지만 수정 버전에 대한 독점권을 인정하는 라이선스를 갖고 있기 때문에 많은 컴퓨터 회사가 이 프로그램을 채택하게 되었다.그들은 X 윈도우 시스템을 그들이 독점적으로 판매하고 있는 유닉스 시스템에 바이너리 형태로만 추가한 뒤에 기존의 유닉스와 동일한 비공개 계약 조건에 의해서 관리되도록 만들어 버렸다. 이러한 종류의 X 윈도우 시스템은 유닉스와 마찬가지로 더 이상 자유 소프트웨어가 아닌 것이다.

X 윈도우 시스템의 개발자들은 이러한 결과가 잘못된 것이라고 생각하지 않았으며 오히려 그렇게 되기를 의도하고 있었다. 왜냐하면 그들의 목적은 '자유'가 아니라 많은 사용자를 가진 프로그램을 만드는 데 '성공'하는 것이었기 때문이다. 그들은 사용자들이 프로그램에 대한 자유를 갖게 되는가의 여부에 상관 없이 오직 많은 사용자를 갖게 되는 데만 관심을 두었다.

이러한 그들의 선택은 결국 "이 프로그램은 자유로운가?"라는 질문에 대한 답을 구할 때 어떤 기준에 의해서 '자유'의 총량을 산출할 것인가라는 역설적인 상황을 초래하게 만들었다. 만약 MIT의 배포 기준에 따라서 자유를 판단한다면 X 윈도우 시스템은 자유 소프트웨어라고 할 수 있지만, 일반적인 사용자에게 허용된 자유를 생각한다면 X 윈도우 시스템은 독점 소프트웨어라고 할 수 있다. X 윈도우를 사용하는 사용자 대부분은 유닉스와 함께 제공되는 독점 버전을 사용하며, 이러한 버전은 자유롭게 사용할 수 있는 것이 아니다.

카피레프트와 GNU GPL

GNU의 목적은 GNU를 널리 알리는 것이 아니라 사용자들에게 자유를 주는 것이다. 따라서 우리의 배포 기준에는 GNU 소프트웨어가 독점 소프트웨어로 변질되는 것을 막을 수 있는 조항이 필요했으며, 이를 위해서 '카피레프트copyleft'[08]라는 방식을 사용했다.

카피레프트는 저작권법을 근간으로 하지만 저작권법이 가진 주된 목적을 반대로 이용해서 소프트웨어를 개인의 소유로 사유화시키는 대신 자유로운 상태로 유지시키는 수단으로 삼는 것이다.

카피레프트의 핵심은 프로그램을 실행하고 복제할 수 있는 권리와 함께 개작된 프로그램에 대한 배포상의 제한 조건을 별도로 설정하지 않는 한, 개작과 배포에 대한 권리 또한 모든사람에게 허용하는 것이다. 즉, 프로그램에 대한 실행과 복제, 개작, 배포의 모든 자유를 허용하는 것이다. 이러한 방법을 통해서 '자유 소프트웨어'라는 용어의 핵심인 '자유'를 모든 사람에게 보장할 수 있고, 프로그램을 입수한 사람은 그 누구도 뺏을 수 없는 권리를 갖게 된다.

카피레프트가 실제로 유효하려면 개작된 프로그램 또한 자유로워야만 한다. 만약 우리가 만든 프로그램에 기반한 2차적 프로그램이 만들어진다면, 이러한 보장이

있어야만 우리 모두가 개작된 프로그램을 사용할 수 있을 것이다. 직업 프로그래머가 GNU 소프트웨어를 개량하기 위해서 자원했을 경우에도 카피레프트는 그들의 고용주가 개작된 프로그램을 독점 소프트웨어로 만들려는 것을 방지할 수 있다.

프로그램을 사용하는 모든 사람의 자유를 확실히 보장하려면 개작된 부분이 자유로워야 한다는 요건은 매우 본질적인 것이다. X 윈도우 시스템을 독점 소프트웨어의 형태로 만든 기업은 그들의 운영체제와 하드웨어에 X 윈도우를 이식하기 위해서 어느 정도 수정을 가했는데, 그들이 수정한 부분은 X 윈도우 전체와 비교하면 작은 부분에 불과하지만 하찮은 것은 아니다. 그러나 단지 프로그램을 수정했다는 이유만으로 사용자들의 자유를 제한할 수 있다면 누구나 이런 방법을 악용해서 쉽게 이익을 얻을 수 있을 것이다.

위에서 언급한 문제는 자유 프로그램과 그렇지 않은 프로그램을 함께 결합하는 경우와도 연관이 있다. 이러한 결합으로 이루어진 결과물은 필연적으로 자유 소프트웨어가 될 수 없으며, 자유 소프트웨어가 아닌 프로그램이 갖고 있던 자유의 결여가 결국 결과물 전체로 확대되어 버리게 된다. 프로그램 사이의 이러한 결합을 허용하는 것은 배를 침몰시키기에 충분한 구멍을 만드는 것과 같으므로 카피레프트가 가져야 할 가장 중요한 요건은 바로 이러한 구멍을 막는 것이다. 즉, 카피레프트로 배포된 프로그램에 어떠한 첨삭이 일어나거나 다른 프로그램과 함께 결합된다고 하더라도 그 결과물에는 반드시 카피레프트가 적용되어야 하는 것이다.

대부분의 GNU 소프트웨어는 카피레프트를 실제로 구현한 라이선스 기준인 GNU General Public License[09]가 사용된다. 우리는 상황에 따라서 다른 종류의 카피레프트를 사용하기도 하는데, GNU 매뉴얼의 경우에는 GNU GPL의 엄밀한 조건이 모두 필요하지 않기 때문에 좀 더 간단한 형태의 카피레프트를 사용한다.

자유 소프트웨어 재단

Emacs의 사용에 대한 관심이 증가하면서 사람들이 GNU 프로젝트에 참여하기 시작했고, 우리는 지금이 기금을 모을 적절한 시기라는 결정을 하게 되었다. 그래서 1985년에 자유 소프트웨어의 개발을 위해서 면세 혜택이 주어지는 자유 소프트웨어 재단Free Software Foundation, FSF을 설립하게 된다.

자유 소프트웨어 재단, 즉 FSF는 Emacs의 테이프 배포 사업을 맡게 되었고 점진적으로 다른 종류의 자유 소프트웨어까지 테이프에 담아서 판매했는데, 여기에는 GNU 이외의 자유 소프트웨어도 포함되었고 매뉴얼의 판매 또한 병행되었다.

FSF는 기부를 받지만 대부분의 운영 자금은 자유 소프트웨어의 판매와 이와 관련된 부수적인 서비스를 통해서 충당된다. 현재 FSF는 소스 코드나 바이너리가 담겨 있는 CD-ROM과 인쇄물로 만들어진 매뉴얼(매뉴얼도 카피레프트에 의해서 관리되므로 개작과 재배포가 허용된다) 그리고 주문자가 선택한 플랫폼에 맞춰서 모든 소프트웨어를 컴파일해서 수록한 디럭스 배포판Deluxe Distribution을 판매한다.

자유 소프트웨어 재단이 고용한 사람들은 그동안 많은 종류의 GNU 소프트웨어 패키지를 만들거나 관리해왔는데, 이 중에서 특히 주목할 만한 것은 C 라이브러리와 셸이다. GNU C 라이브러리는 GNU/Linux 시스템에서 실행되는 모든 프로그램이 리눅스와 연동하기 위해서 반드시 필요한 것이다. C 라이브러리는 자유 소프트웨어 재단의 스탭 중 한 사람인 롤랜드 맥그래스Roland McGrath가 개발한 것이며, 대부분의 GNU/Linux 시스템에서 사용하는 셸 프로그램인 배쉬BASH는 FSF의 고용자였던 브라이언 폭스Brian Fox가 개발한 한 것이다(참고로 BASH는 Bourne Again SHell[10]의 약자로 사용된 것이다).

우리는 이러한 프로그램을 개발하기 위해서 기금을 사용했는데, 이는 GNU 프로젝트가 단순히 특정한 작업 도구나 개발 환경에 국한된 것이 아니기 때문이다. 우

리의 목표는 완성된 형태의 운영체제를 만드는 것이며 이러한 프로그램은 우리의 목표를 위해서 필요한 것들이었다.

자유 소프트웨어에 대한 지원

자유 소프트웨어가 가진 철학은 널리 만연되어 있는 특정한 형태의 상업적 관행에 반대하지만 그렇다고 상업 행위 자체를 거부하는 것은 아니다. 사용자의 자유를 존중하는 사업이라면 우리는 그들의 성공을 기원할 것이다.

Emacs의 판매는 자유 소프트웨어 사업의 한 가지 예라고 할 수 있다. Emacs의 판매 사업을 자유 소프트웨어 재단으로 이관한 뒤에 나는 또 다른 생계 수단을 강구해야만 했는데, 나는 내가 개발했던 자유 소프트웨어와 관련된 서비스를 제공하는 방법을 생각하게 되었다. 여기에는 GNU Emacs를 프로그래밍하거나 GCC를 최적화시키는 방법에 대한 교육과 GCC를 새로운 플랫폼으로 이식하는 작업이 주가 되는 소프트웨어 개발이 포함되었다.

오늘날에는 자유 소프트웨어와 관련된 이러한 종류의 사업이 많은 기업에 의해서 이루어지고 있다. 어떤 업체는 자유 소프트웨어로 구성된 CD-ROM을 판매하기도 하고, 또 다른 업체들은 사용자들의 질문에 답변해 주는 것부터 시작해서 버그를 잡거나 프로그램에 새로운 주요 기능을 추가하는 등의 다양한 수준과 요구에 맞는 지원을 유료로 제공하기도 한다. 한 단계 더 나아가서 이제는 새로운 자유 소프트웨어 제품을 기반으로 사업을 시작하는 기업들도 생겨나고 있다.

그러나 많은 기업이 실제로는 단지 자유 소프트웨어와 함께 연동할 수 있는 소프트웨어를 기반으로 한 업체일 뿐임에도 자신을 '오픈 소스'라는 이름과 연관시키고 있다는 사실에 각별히 주의해야 한다. 이러한 기업은 자유 소프트웨어 업체가 아니라 그들의 상품을 이용해서 사용자가 자유로부터 멀어지도록 유혹하는 독점 소프트웨어 업체인 것이다. 그들의 제품이 가진 유혹은 자유보다는 편리함인데, 그들은

이를 '부가가치(附加價値)'라는 이름으로 우리가 수용하길 요구한다. 그러나 자유라는 측면에서 이를 판단한다면 우리는 이러한 제품을 부가가치가 아닌 '자유감치(自由減値)' 제품이라고 불러야 마땅할 것이다.

기술적인 목표

GNU의 주된 목표는 자유 소프트웨어가 되는 것이다. GNU가 유닉스에 비해서 기술적인 장점을 전혀 갖지 못하게 되더라도, 사용자가 서로 협력할 수 있게 하면 사회적인 이점을 갖게 될 것이고, 사용자의 자유를 존중하게 된다는 점에서 윤리적인 이점도 있다고 할 수 있다.

그러나 우수한 기술의 표준을 GNU에 수용했던 것은 무척이나 당연한 일이었다. 예를 들어 제한된 영역의 메모리 사용을 극복하기 위해서 동적 할당 방식의 자료구조를 도입했으며, 8비트 코드를 처리할 수 있도록 해서 어떠한 형태의 입력에 대해서도 이를 정상적으로 처리할 수 있게 했다.

또한 16비트 컴퓨터를 지원하지 않도록 결정함으로써 유닉스에서 채택한 작은 크기의 메모리로 인한 문제를 없앨 수 있게 되어 1MB가 넘지 않는 한도에서는 메모리의 사용을 줄이기 위해서 노력할 필요가 없어졌다(GNU 시스템이 완성될 시점에는 32비트 컴퓨터가 일반적인 추세가 되리라는 것이 확실했기 때문이다). 따라서 프로그램에서 큰 용량의 파일을 처리하는 것이 수월해졌기 때문에 프로그래머에게 입출력에 대한 문제를 고려하지 말고 입력 파일 전체를 메모리로 올려서 사용하도록 권장할 수 있었다.

결국 이러한 결정은 GNU 프로그램이 같은 유닉스 프로그램에 비해서 향상된 속도와 신뢰성을 갖도록 해주었다.

기증받은 컴퓨터

GNU 프로젝트의 명성이 높아지면서 유닉스가 탑재된 시스템을 기증하는 사람들이 생겨나기 시작했다. GNU를 구성하는 프로그램을 개발하는 가장 손쉬운 방법은 유닉스 환경에서 먼저 개발한 뒤에 이를 하나씩 GNU에 맞게 수정하는 것이었기 때문에 이러한 시스템은 우리에게 매우 유용한 것이었다. 그러나 그들은 우리가 유닉스를 사용하는 것이 과연 정당한가라는 윤리적인 문제를 제기해왔다. 유닉스는 예나 지금이나 독점 소프트웨어에 해당한다. 그리고 GNU 프로젝트의 철학은 독점 소프트웨어를 사용하지 않겠다는 데 있다. 그런데 독점 소프트웨어를 사용해서 자유 소프트웨어를 만든다는 것은 모순이라는 것이 그들의 주장이었던 것이다.

그러나 자기 자신을 보호하기 위해서 사용한 무력을 정당방위로 인정받을 수 있는 것처럼 다른 사람이 독점 소프트웨어를 사용하지 않을 수 있도록 돕기 위해서 자유 소프트웨어를 만드는 과정에 독점 소프트웨어를 사용할 필요가 있다면 나는 이 또한 정당화될 수 있다는 결론을 내렸다.

그러나 이러한 방법이 정당화될 수 있더라도 독점 소프트웨어를 사용하는 것은 여전히 해악한 것이었다. 그래서 어느 시점에 이르렀을 때, 우리는 더 이상 독점 소프트웨어를 사용하지 않기로 결정하고 기증받은 컴퓨터에서 사용할 수 있는 자유 운영체제를 찾기 시작했다. 사용 가능한 운영체제를 발견한 경우에는 운영체제를 교체했지만, 자유 운영체제를 사용할 수 없는 경우에는 자유 운영체제를 사용할 수 있는 시스템으로 컴퓨터 자체를 교체했다. 오늘날에는 이미 유닉스를 대체할 수 있는 자유 운영체제를 갖고 있기 때문에 유닉스가 설치된 시스템을 전혀 사용하지 않고 있다.

GNU 태스크 리스트

GNU 프로젝트가 진행됨에 따라서, 그리고 발견하거나 개발된 자유 운영체제의 구성 요소들이 많아짐에 따라서 마침내 남아 있는 부분에 대한 목록을 만드는 것이 필요한 시점이 되었다.

우리는 이 목록을 이용해서 남아 있는 부분을 만드는 데 필요한 개발자를 모집했는데, 이 목록은 'GNU 태스크 리스트GNU Task List'라는 이름으로 불리게 되었다. 여기에는 아직 개발되지 않은 유닉스 시스템의 구성 요소뿐만 아니라 하나의 완벽한 운영체제가 되기 위해서 필요하다고 판단한 다양한 종류의 소프트웨어와 문서 프로젝트 등을 이 목록에 추가시켰다.

유닉스와 관련된 구성 요소 대부분은 이제 GNU 태스크 리스트에 남아 있지 않다(대부분의 작업을 모두 완료했으며 자잘한 몇 부분만이 남아 있는 상태다). 그러나 이 목록에는'애플리케이션'이라고 부를 수 있는, 소프트웨어를 만들기 위한 많은 양의 프로젝트가 가득 차 있다. 협소한 사용자 층을 가진 프로그램이 아니라면 대부분의 프로그램은 운영체제와 함께 구성될 만한 유용한 프로그램이라고 할 수 있을 것이다.

유닉스에는 게임 프로그램이 포함되어 있으므로 자연스럽게 GNU에도 게임이 포함되어야 했다. 따라서 GNU 태스크 리스트에는 이러한 게임 프로그램도 처음부터 함께 포함했다. 그러나 게임 프로그램에는 호환성이라는 문제를 고려할 필요가 없기 때문에 유닉스에서 제공하던 게임의 종류를 그대로 따르지 않고 사용자가 좋아할 만한 다양한 종류의 게임을 포함시켰다.

GNU Library GPL

GNU C 라이브러리에는 'GNU Library GPL'이라고 불리는 특별한 종류의 카피레프트가 적용된다. 이는 독점 소프트웨어와의 링크를 허용하는 것으로 이제는

'GNU Lesser GPL'이라는 이름으로 변경되었으며 LGPL이라고 줄여서 부르기도 한다. 그렇다면 왜 이러한 예외를 만들었는가?

LGPL은 우리의 원칙을 수정한 것이 아니다. 우리의 원칙에는 결코 독점 소프트웨어 제품이 우리가 만든 코드를 사용하도록 허용하지 않고 있다(소프트웨어를 우리와 함께 공유하지 않을 것이 명백한 프로젝트에 도움을 줄 이유가 없지 않은가?). C 라이브러리나 다른 라이브러리를 LGPL로 만든 것은 원칙을 수정한 것이 아니라 일종의 전략이라고 할 수 있다.

C 라이브러리는 매우 일반적인 기능을 제공하는 것이어서 모든 독점 운영체제와 컴파일러에는 C 라이브러리가 함께 제공된다. 따라서 GNU C 라이브러리를 자유 소프트웨어에서만 사용할 수 있도록 한다면, 자유 소프트웨어는 별다른 비교 우위를 갖게 되지 못할 것이다. 즉, GNU C 라이브러리를 사용하려는 의욕을 저하시키게 되는 것이다.

GNU/Linux를 포함한 GNU 시스템에서는 오직 GNU C 라이브러리만을 사용할 수 있기 때문에 이러한 문제에서 예외일 수 있다. 따라서 GNU C 라이브러리의 배포 조건에 고려되어야 할 점은 독점 소프트웨어가 GNU 시스템용으로 컴파일할 수 있도록 GNU C 라이브러리의 사용을 허용할 것인가라는 문제로 귀착된다.

독점 소프트웨어를 GNU에서 사용할 수 있도록 허용하는 것에는 어떠한 윤리적인 문제도 없다. 그러나 전략적으로 볼 때 이를 허용하지 않는 것은 자유 소프트웨어의 개발을 촉진한다기보다 오히려 GNU 시스템의 사용을 저하시킬 가능성이 크다. 이것이 GNU C 라이브러리를 LGPL로 관리하는 전략적인 이유다. 다른 라이브러리는 상황에 맞는 전략적 판단을 통해서 결정하는 것이 필요할 것이다.

만약 어떠한 라이브러리가 특정한 종류의 프로그램을 개발하는 데 중요한 역할을 수행하게 된다면, 이러한 라이브러리는 GPL로 관리해서 자유 소프트웨어에서만

사용하도록 제한하는 것이 바람직하다. 그러면 자유 소프트웨어의 개발자는 독점 소프트웨어 개발자에 대한 비교 우위와 이점을 갖게 될 수 있다.

GNU Readline의 예를 보자. 이 라이브러리는 BASH의 명령행 편집 기능을 구현하기 위해서 개발된 것이다. Readline은 LGPL이 아닌 GPL에 의해서 관리되었기 때문에 아마도 LGPL로 배포했을 경우보다는 그 사용 빈도가 많지 않았을 것이다. 그러나 이로 인해서 우리가 잃은 것은 아무것도 없다. 반면에 Readline을 채택한 유용한 자유 소프트웨어가 하나라도 만들어 졌다면 이는 공동체 전체에 있어서 매우 유용한 수확이라고 할 수 있다.

독점 소프트웨어의 개발자들은 자본이라는 우위를 갖고 있다. 그러나 자유 소프트웨어의 개발자들은 서로를 위해서 이점을 만들어 주어야 한다. 나는 언젠가는 우리가 독점 소프트웨어에 라이선스를 허용하지 않는 넉넉한 수의 GPL 라이브러리를 갖게 되기를 희망한다. 이러한 라이브러리들은 새로운 자유 소프트웨어 안에서 유용한 기능을 수행하는 모듈로 활용될 것이며, 좀 더 나은 자유 소프트웨어를 개발하는 데 있어서 중요한 이점을 제공하게 될 것이다.

가려운 곳을 긁는다?

에릭 레이먼드Eric Raymond는 "소프트웨어에 있어서 모든 좋은 성과 들은 개발자 자신의 가려운 곳을 긁는 것으로부터 시작된다."[11]라고 주장한다. 때때로 이는 타당한 말일 것이다. 그러나 GNU 소프트웨어의 핵심 부분은 대부분 완전한 자유 운영체제를 만들기 위한 목적으로 개발된 것이다. GNU는 가려울 때마다 가려운 곳을 긁듯이 그때그때 만들어진 것이 아니라 비전과 계획에 의해서 만들어진 것이다. 예를 들어 우리가 GNU C 라이브러리를 개발한 이유는 유닉스 형태의 운영 환경에서 C 라이브러리가 필요했기 때문이고, BASH를 개발한 이유 또한 유닉스 환경이 셸을 요구하기 때문이었다. GNU tar나 내가 직접 개발한 GNU C 컴파일러와 GNU Emacs, 그리고 GDB와 GNU Make도 같은 이유에서 개발한 것들이다.

어떠한 종류의 GNU 프로그램은 자유에 대한 위협에 대처하기 위해서 만들어진 것이다. gzip은 LZW 압축 알고리즘의 특허 문제로 인해서 공동체 안에서 사라져 버린 압축 프로그램을 대체하기 위해서 개발되었다. 우리는 독점 라이브러리로 인해서 일어나는 문제들을 해결하기 위해서 LessTif를 개발할 사람들을 모집했으며, 더욱 최근에는 GNOME과 Harmony를 만들기 시작했다. 또한 사생활과 자유 중에서 하나를 선택해야만 하는 불합리한 문제를 없애기 위해서 자유 소프트웨어가 아닌 기존의 인기 있는 암호화 소프트웨어를 대체할 GNU Privacy Guard를 개발하고 있기도 하다.

물론 이러한 프로그램을 개발하는 개발자들은 자신의 작업에 흥미를 갖고 있으며, 많은 사람이 그들의 필요와 관심에 따라서 다양한 기능을 덧붙여 가고 있다. 그러나 이러한 것이 프로그램이 존재하는 이유는 아니다.

예기치않은개발들

GNU 프로젝트를 시작할 당시에는 GNU 시스템을 모두 완성한 뒤에 완성된 시스템 전체를 공개하려고 생각했었다. 그러나 다음과 같은 이유로 인해서 그렇게 되지 못했다.

GNU 시스템을 구성하는 각각의 요소는 유닉스에서 먼저 구현한 뒤에 이를 GNU로 옮긴 것이기 때문에 GNU 시스템이 완성되기 오래전부터 이미 유닉스에서 사용되었다. 이러한 프로그램 중에서 어떠한 것들은 유명해졌으며, 사용자가 프로그램을 확장하거나 호환되지 않던 다양한 종류의 유닉스로 이식시켰다. 또한 유닉스 이외의 운영체제로 이식이 이루어지기도 했다.

이러한 과정을 통해서 해당 프로그램은 좀 더 강력한 기능을 갖게 되었고 기부와 공헌자를 GNU 프로젝트로 끌어들이게 했다. 그러나 이는 GNU 개발자들로 하여금 GNU 시스템을 완성시켜 나가는 것보다 오히려 기존의 구성 요소에 새로운 기

능을 덧붙이거나 다른 시스템으로 이식하는 데 시간을 투자하게 만들었다. 결국 동작 가능한 최소한의 시스템을 완성하는 데 몇 년이 지연되는 결과를 가져왔다.

GNU HURD

1990년 무렵에는 GNU 시스템이 거의 완성되었지만, 운영체제를 구성하는 핵심 부분 중의 하나인 커널이 누락된 상태였다. 우리는 Mach(마하 또는 마크)를 기반으로 한 서버 프로세스를 연결해서 커널을 구현하기로 결정했다. Mach는 카네기 멜론 대학Carnegie Mellon University에서 개발한 마이크로커널microkernel로 후에 유타 대학University of Utah에서 개발을 이어갔다. 커널은 운영체제를 구성하는 핵심 부분을 가리키는 말인데, 마이크로커널 아키텍처에서는 IPC와 메모리 관리 부분만을 커널에 내장시키고 나머지 부분들은 자율적인 프로세스로 구현한 뒤에 이들을 연결하는 구조를 갖고 있다.

따라서 우리가 만들려고 했던 GNU HURD(허드)는 Mach와 자율적인 프로세스 서버의 집합이라고 할 수 있으며, 유닉스 커널이 가진 다양한 기능을 그대로 수행할 수 있는 것이었다. 그러나 GNU HURD의 개발은 Mach를 자유 소프트웨어로 배포한다고 예고했던 시기까지 지연되었다.

마이크로커널을 채택한 이유의 하나는 소스 코드 차원의 디버거 없이는 가장 힘든 작업이 될 수밖에 없는 커널 프로그램의 디버깅을 피하기 위해서였다. 우리는 완성된 Mach를 활용하면 이러한 작업을 생략할 수 있으며 HURD 서버를 디버깅하는 데는 GDB를 사용할 수 있으리라고 생각했다. 그러나 이러한 작업이 가능하기 위해서는 상당한 기간이 필요했으며, 상호 메시지 전달을 위한 멀티 스레드 서버를 디버깅하는 것도 매우 어려운 작업이었다. 이러한 이유 때문에 HURD를 만드는 작업에는 많은 시간이 소요되었다.

Alix

GNU 커널의 이름을 처음부터 HURD라고 부르려 했던 것은 아니다. 원래의 이름은 당시 내 연인의 이름을 따서 Alix라고 했었는데, 유닉스 시스템 관리자였던 그녀는 Alix라는 이름이 유닉스의 버전마다 관례처럼 붙여지던 별칭과 잘 어울린다면서 "누군가 내 이름을 따서 커널 이름을 지어야 할 걸."이라고 친구들에게 농담처럼 말하곤 했다. 나는 아무 말도 하지 않았지만 Alix를 커널의 이름으로 사용해서 그녀를 놀라게 해주리라 마음먹었다.

그러나 이러한 나의 생각은 실현되지 못했다. 커널의 주요 개발자였던 마이클 부쉬넬Michael Bushnell(현재의 이름은 토마스다)이 HURD라는 이름을 더 선호했기 때문이다. 또한 그는 Alix를 커널의 이름에서 HURD 서버로 메시지를 전송하는 방법을 통해서 시스템 콜을 트랩하고 제어할 수 있는 커널의 한 부분으로 변경시켰다.

훗날 Alix와 나는 헤어지게 되었고 그녀는 그녀의 이름을 바꿨다. 또한 이러한 사연과는 무관하게 HURD의 디자인도 C 라이브러리가 서버로 메시지를 직접 전달할 수 있는 형태로 수정되었다. 이러한 이유로 Alix는 HURD에서 사라졌다.

그러나 이러한 일이 일어나기 전에 Alix의 친구 중 한 명이 HURD의 소스 코드에 포함되어 있던 Alix라는 이름을 보게 되었고 Alix에게 그 사실을 전해주어 나의 의도는 그녀에게 전달될 수 있었다.

리눅스와 GNU/Linux

현재까지도 GNU HURD는 하나의 제품으로 사용될 수 있을 만한 상태가 아니다. 그러나 다행스럽게도 사용 가능한 또 다른 커널이 있었는데, 1991년에 리누스 토발스Linus Torvalds가 유닉스 커널과 호환 가능하게 만든 리눅스라는 이름의 커널이 그것이었다. 1992년 무렵에 우리는 완전하지 않았던 GNU 시스템과 리눅스를 결합함으로써 하나의 완성된 자유 운영체제를 만들 수 있었다(이들을 결합한다는 것

은 물론 단순한 작업이 아닌 상당한 노력이 필요한 실제적인 작업이었다). 따라서 현재 사용되는 GNU 시스템은 리눅스 덕분에 가능했던 것이라고 할 수 있다. 우리는 이 시스템을 GNU/Linux라고 부르는데, 이는 리눅스를 시스템 커널로 채용한 GNU 시스템을 지칭하는 것이다.

미래의 도전들

우리는 다양한 종류의 자유 소프트웨어를 개발함으로써 우리의 역량을 증명해왔다. 그러나 이것이 우리에게 장애가 없는 무소불위의 능력이 있음을 의미하지는 않는다. 몇 가지 장애물이 자유 소프트웨어의 앞날을 불확실하게 만들고 있기 때문이다. 이러한 장애를 극복하기 위해서는 끊임 없는 노력과 인내가 필요하며 때때로 몇 년이 소요되기도 할 것이다. 또한 자유를 소중히 여기고 이를 누구에게도 빼앗기지 않겠다는 의사를 분명히 표현하려는 사람들의 결단이 필요하다.

우리에게 당면해 있는 장애는 다음과 같은 것이다.

비밀스런 하드웨어

하드웨어 제조업에서는 하드웨어 스펙을 비밀로 하는 경우가 점점 더 늘어나고 있다. 이 때문에 자유롭게 사용할 수 있는 드라이버를 만드는 것이 힘들어지고 결과적으로 리눅스와 X Free 86에서 새로운 하드웨어를 지원하는 것이 어려워지고 있다. 현재 우리들은 완성된 자유 운영체제를 갖고 있지만 앞으로 개발될 컴퓨터를 지원할 수 없다면 그렇게 되지 못할 수도 있을 것이다.

이러한 문제에 대처할 수 있는 방법은 두 가지가 있다. 첫 번째 방법은 리버스 엔지니어링reverse engineering을 채용하는 것이고, 두 번째 방법은 사용자들이 결과를 지원하는 것이다. 리버스 엔지니어링이란 이미 생산되는 하드웨어를 분석해서 설계 구조와 스펙 등을 알아내는 방법이다. 따라서 프로그래머가 리버스 엔지니어링을 수행하면 일반 사용자는 그 결과를 통해서 자유 소프트웨어에서 사용할 수 있는 하

드웨어를 선택할 수 있다. 그렇게 되면 자유 소프트웨어를 사용하는 사람이 증가할수록 하드웨어 스펙을 공개하지 않는 방침은 결국 스스로의 화를 자초하는 선택이 될 것이다.

리버스 엔지니어링은 매우 방대한 작업이다. 우리가 과연 이러한 일을 수행할 수 있는 결단을 가진 프로그래머를 가질 수 있을까? 이는 우리가 자유 소프트웨어가 아닌 드라이버를 과감하게 포기하고 자유 소프트웨어라는 원칙을 단호하게 지켜나간다면 충분히 가능하리라고 생각한다. 우리 중 많은 사람이 자신이 가진 여분의 시간과 돈을 투자한다면 자유롭게 사용할 수 있는 드라이버를 갖지 못할 이유는 없지 않을까? 자유를 누리려는 결단이 널리 확산된다면 충분히 가능할 것이다.

자유 소프트웨어가 아닌 라이브러리

자유 운영체제에서 사용할 수 있는 자유 소프트웨어가 아닌 라이브러리는 자유 소프트웨어 개발자에게 있어 일종의 함정과 같다. 이러한 라이브러리가 가진 매혹적인 기능을 외면한다는 것은 개발자에게 있어 매우 어려운 일이다. 그러나 만약 이러한 유혹에 넘어간다면 그것은 스스로 자신의 무덤을 파는 것과 같다. 왜냐하면 이러한 라이브러리를 이용해서 만들어진 프로그램은 자유 운영체제에 포함될 수 없기 때문이다(좀 더 정확하게 말하면 프로그램은 포함시킬 수 있겠지만 라이브러리 자체가 포함될 수 없기 때문에 결국 이러한 프로그램은 실행될 수 없는 것이다). 더욱 심각한 경우는 독점 라이브러리를 사용한 프로그램이 매우 인기 있는 프로그램이 되었을 경우인데 이러한 라이브러리에 관심을 갖고 있지 않던 프로그래머를 같은 유혹에 빠뜨릴 수 있다는 점이다.

이러한 경우의 첫 번째 실례는 1980년대에 사용되던 모티프Motif 툴킷에서 찾아볼 수 있다. 그 당시에는 아직 자유 운영체제가 존재하지 않았지만 Motif가 갖고 있던 문제가 어떤 결과를 초래하리라는 것은 이미 자명한 일이었다. GNU 프로젝트는 두 가지 방법으로 여기에 대응했다. 첫째는 자유 소프트웨어 프로젝트들가 Motif

뿐만 아니라 자유 소프트웨어에 해당하는 X 툴킷 위젯도 지원하도록 권고한 것이고, 또 하나는 Motif를 대체할 수 있는 라이브러리의 개발을 추진한 것이다. 이 작업은 몇 년이 소요되었는데, 열성적인 프로그래머들 덕분에 1997년에 이르러 Motif를 사용하는 대부분의 프로그램을 지원할 수 있는 LessTif 라이브러리가 완성되었다.

1996년과 1998년 사이에는 이른바 Qt라고 불리는 또 다른 GUI 툴킷 라이브러리가 등장했다. Qt는 데스크톱 환경의 자유 소프트웨어인 KDE에 사용되는 핵심 요소다. 그러나 자유 운영체제인 GNU/Linux 시스템에서는 Qt의 라이선스와 관련해서 KDE를 사용할 수 없었다. GNU/Linux를 판매하는 일부 상용 배포판 업체에서는 KDE를 함께 패키징해서 활용성을 증대시킨 시스템을 만들기도 했지만 결국 이것은 자유를 감소시키는 것이었다. KDE 그룹은 좀 더 많은 프로그래머가 Qt를 사용하도록 적극 권장했고, 새롭게 리눅스를 사용하게 된 수많은 사람은 이러한 문제를 미처 생각하지도 못한채 KDE를 사용하게 되었다. 상황이 무척 심각해진 것이다. 자유 소프트웨어 공동체는 GNOME과 Harmony라는 두 가지 방법으로 이 문제에 대응했다.

우리는 먼저 GNU Network Object Model Environment[12], 즉 GNOME(그놈)이라고 불리는 데스크톱 환경의 개발을 시작했다. 이 프로젝트는 미구엘 드이카자[13]에 의해서 1997년부터 시작되었고 레드햇 소프트웨어가 이를 후원한다. GNOME은 데스크톱 환경과 유사한 기능을 제공하지만 오직 자유 소프트웨어만으로 만들어진다. 또한 C++ 이외에 다양한 종류의 언어를 지원함으로써 기술적인 장점도 함께 제공한다. 그러나 GNOME의 주된 목적은 자유라고 할 수 있다. GNOME 환경은 자유 소프트웨어가 아닌 어떠한 프로그램의 기능도 자유 소프트웨어 안에서 충족시키기 위한 것이다. 또 하나의 대안인 Harmony는 KDE 소프트웨어들을 Qt 없이도 사용할 수 있게 하기 위해서 개발된 라이브러리다.

1998년 11월에 Qt 개발자들은 Qt를 자유 소프트웨어로 변경할 것이라고 발표했다. 그러나 이는 아직까지 확실한 것이 아니며, 부분적으로 Qt가 자유 소프트웨어가 아니었을 때 가졌던 문제에 우리가 단호하게 대처함으로써 나타난 결과라고 할 수 있다(Qt의 새로운 라이선스에도 여전히 불충분한 점이 많으므로 가능한 Qt를 사용하지 않는 것이 바람직하다).

자유 소프트웨어가 아닌 또 다른 라이브러리의 유혹에 우리는 어떻게 대응해야 할까? 공동체 일원이라면 모두가 이러한 유혹으로부터 벗어날 필요성에 공감하게 될까? 그렇지 않다면 편리함을 핑계삼아 자유를 포기하고 중요한 문제를 만들어 내지는 않을까? 우리의 미래는 다름 아닌 우리 자신의 철학에 달려 있는 것이다.

소프트웨어에 대한 기술 특허

우리가 당면한 가장 큰 위협은 바로 소프트웨어에 대한 특허에서 발생된다. 알고리즘과 새로운 기술에 부여되는 특허는 20년까지 보장되므로 이러한 영역에는 자유 소프트웨어가 접근할 수 없는 것이다. LZW 압축 알고리즘에 대한 기술 특허는 1983년에 신청되었다. 그리고 아직까지도 우리는 정상적인 GIF 압축을 생성할 수 있는 자유 소프트웨어를 공개하지 못하고 있다. 또한 1998년에는 특허권 침해에 대한 소송 위협 때문에 배포판에서 MP3 오디오 압축 프로그램을 제외해야 했다.

특허권에 대처할 수 있는 몇 가지 방법은 특정한 특허가 특허로 인정될 수 없다는 반대 증거를 찾거나 특허가 취득된 기술을 대체할 수 있는 방법을 찾는 것이다. 그러나 이러한 방법이 항상 적용될 수 있는 것은 아니며, 이러한 방법이 모두 실패할 경우에는 사용자가 원하는 기능을 자유 소프트웨어에 포함시킬 수 없게 될 것이다. 만약 이러한 일이 일어난다면 어떻게 해야 할 것인가?

자유 소프트웨어가 가진 자유의 가치를 중요하게 생각하는 사람은 어떠한 상황에서도 자유 소프트웨어를 버리지 않을 것이다. 우리는 특허와 관련된 기능이 없더라

도 나름대로 원하는 작업을 해 나갈 수 있을 것이다. 그러나 자유 소프트웨어가 갖게 될 기술적 우위를 기대하고 이를 사용했던 사람은 특허에 의해서 기술적 장점이 사라진 이후에는 자유 소프트웨어가 실패했다고 단언할 것이다. 따라서 '성당' 형태의 개발 모델이 가진 실제적인 효과와 특정한 자유 소프트웨어가 가진 안정성과 성능에 대해서 말하는 것이 사람들에게 유용한 동안에는 단지 그러한 말만을 할 것이 아니라 자유와 자유를 지켜나가기 위한 원칙도 함께 이야기해야만 한다.

자유 문서

우리의 자유 운영체제가 안고 있는 가장 절실한 문제는 소프트웨어에 대한 부분이 아니라 운영체제에 함께 포함시킬 양질의 문서가 부족하다는 점이다. 문서 자료는 모든 소프트웨어에 있어서 필수적인 부분이다. 만약 핵심적인 자유 소프트웨어가 자유롭게 이용할 수 있는 좋은 문서 자료 없이 제공된다면 이는 심각한 문제라고 할 수 있는데, 현재 우리는 이러한 문제를 많이 갖고 있다.

자유 소프트웨어와 마찬가지로 자유 문서에는 가격이 아닌 자유가 중요한 관건이 된다. 자유 문서의 판단 기준은 자유 소프트웨어와 거의 같다고 할 수 있다. 즉, 모든 사용자에게 자유를 허용하는가다. 상업적 판매를 포함한 재배포가 온라인 또는 서적의 형태로 모두 허용되어야 하며, 이러한 문서 자료가 해당 프로그램과 함께 제공될 수 있어야 한다.

개작에 대한 허용 또한 필수적인 부분이다. 그러나 나는 일반적으로 모든 종류의 글과 책을 개작할 수 있도록 허용해야 한다고 생각하지는 않는다. 예를 들어 여러분이 읽는 바로 이 글처럼 우리의 생각이나 활동을 설명한 글을 개작할 수 있도록 허용할 필요는 없는 것이다.

그러나 자유 소프트웨어를 위해서는 문서에 대한 개작의 허용이 필수적이라고 할 수 있다. 소프트웨어를 개작할 수 있는 권리를 사용해서 소프트웨어를 수정하거나

새로운 기능을 추가했다면 자신의 작업을 책임지고 마무리하기 위해서 기존의 매뉴얼을 변경된 기능에 맞는 좀 더 정확하고 유용한 문서로 고치려고 할 것이다. 그러나 프로그래머에게 이러한 작업이 허용되지 않는다면 결국 공동체가 필요로 하는 요구를 충족하지 못하게 되는 것이다.

문서의 수정 형태에 대한 외형적인 제한 조건을 설정하는 것은 경우에 따라서 별다른 문제가 되지 않는다. 예를 들어 원저작자의 저작권 사항이나 배포 조건 또는 저자들의 명단을 보존해야 한다는 제한이나 문서가 수정되었다는 사실을 명시해야 한다는 제한, 그리고 수정되지 않은 부분은 그대로 남겨 두어야 한다는 등의 제한 조건은 이러한 부분이 기술적인 내용을 담지 않는 한 큰 문제가 되지 않는다. 왜냐하면 이러한 종류의 제한은 프로그래머가 개작된 프로그램에 맞게 매뉴얼을 수정하는 데 별다른 지장을 주지 않기 때문이다. 즉, 이러한 제한은 공동체가 매뉴얼을 온전히 활용하는 것을 제한하지 않기 때문에 문제가 되지 않는다.

그러나 기술적인 내용에 관련된 부분들은 모두 수정할 수 있어야 하며 수정된 문서는 어떠한 배포 매체와 배포 방법을 통해서도 배포될 수 있어야 한다. 그렇지 않다면 이러한 제한은 공동체의 필요를 가로막는 장애가 되어 결국 자유롭게 사용할 수 있는 또 다른 매뉴얼을 필요로 하게 될 것이다.

그렇다면 자유 소프트웨어의 개발자들이 모든 종류의 매뉴얼을 만들 필요성을 인식하고 이를 실천할 수 있을까? 다시 한 번 반복하는 말이지만, 우리의 미래는 다름 아닌 우리들 자신의 철학에 달려 있는 것이다.

우리는 자유에 대해서 이야기해야만 한다

오늘날에는 대략 천만 명 정도의 사람들이 레드햇 리눅스나 데비안 리눅스와 같은 GNU/Linux 시스템을 사용하는 것으로 추산된다. 사람들 대부분은 실용적인 목적에 따라 움직이며, 자유 소프트웨어는 이러한 부분을 하나둘씩 쌓아 왔다.

대중적이라는 측면이 가져오는 긍정적인 결과는 명확한 것이다. 이전에 비해서 자유 소프트웨어의 개발에 많은 관심을 갖게 될 것이며, 자유 소프트웨어 산업에 좀 더 많은 고객층이 형성될 것이다. 또한 독점 소프트웨어 제품을 개발하는 대신 상용 자유 소프트웨어를 개발하도록 기업들을 더욱 많이 장려할 수 있을 것이다.

그러나 자유 소프트웨어의 바탕에 깔려 있는 본질적인 철학적 인식보다 자유 소프트웨어 자체에 대한 관심이 너무도 빨리 증가한다는 것이 우리가 당면해 있는 문제다. 지금까지 네 개의 범주로 나눠서 설명했던 미래에 대한 도전과 위협을 극복해 나갈 수 있는 능력은 자유를 지키겠다는 우리의 단호한 의지에 달려 있다. 우리의 공동체가 이러한 의지를 견지하기 위해서는 우리와 합류하려는 사람들에게 이러한 철학적 생각을 깊이 있게 인식시켜 주어야 한다.

그러나 지금까지는 그렇게 하지 못했다고 볼 수 있다. 공동체의 일원들에게 의식을 심어 주는 것보다 우선 공동체 안으로 끌어들이려는 노력이 우세했던 것이다. 그러나 우리는 두 가지 작업을 함께 병행해야 하며 이러한 노력이 균형을 이루도록 힘써야 한다.

오픈 소스

1998년부터는 새로운 사용자들에게 자유에 대해 이야기하는 것이 더욱 어려워졌다. 왜냐하면 공동체의 일부에서 '자유 소프트웨어'라는 말 대신 '오픈 소스 소프트웨어'라는 표현을 사용하기 시작했기 때문이다.

이 용어를 선호하는 사람들의 의도는 '자유'를 의미하는 영어 단어인 'Free'가 가진 무료(無料)와 자유(自由)라는 의미상의 혼란을 방지하기 위한 것으로 이는 타당한 것이라 할 수 있다. 그러나 또 다른 사람들은 '자유'라는 단어가 함축한 자유 소프트웨어 운동과 GNU 프로젝트의 정신을 의도적으로 퇴색시키기 위해서 '오픈 소스'라는 표현을 사용하며 자유나 공동체보다는 이윤에 더 높은 가치를 부여하는

기업 경영인과 사용자에게 좀 더 친밀하게 다가서기 위해서 이 말을 사용하기도 한다. 따라서 '오픈 소스'라는 용어는 높은 품질과 강력한 성능을 가진 소프트웨어를 만들 수 있다는 가능성에 초점을 맞춘 말이기 때문에 자유와 공동체 그리고 원칙과 같은 개념을 의도적으로 전달하지 않으려는 표현이라고 할 수 있다.

'리눅스 XXX' 등과 같이 리눅스라는 단어가 제호로 들어간 잡지는 분명한 실례의 하나다. 이러한 잡지에는 GNU/Linux에서 사용 가능한 독점 소프트웨어의 광고로 가득 차 있다. Motif나 Qt와 같은 라이브러리가 다시 생겨났을 때, 과연 이러한 잡지가 프로그래머들에게 그 위험성을 경고해줄 것인지 아니면 제품의 광고를 실을 것인지는 명확하지 않은가?

기업이 제공하는 지원은 공동체에 여러 가지 방법으로 도움이 될 수 있지만 그 이외의 것들도 마찬가지로 유용하다. 그러나 자유에 대해서 말하지 않는 방법으로 이러한 지원을 받아낸다면 이는 화를 자초하는 것이라고 할 수 있으며, 좀 더 많은 사용자를 공동체 안으로 끌어들이려는 노력과 공동체의 성원에게 철학적 인식을 심어주려는 우리의 두 가지 목적 안에 존재했던 불균형을 더욱 심화시키는 결과가 될 것이다.

'자유 소프트웨어'와 '오픈 소스'는 같은 부류의 소프트웨어를 가리키는 말이다. 차이점은 소프트웨어와 가치에 대해서 서로 다른 입장을 갖고 있다는 것이다. GNU 프로젝트는 기술적인 측면 그 자체보다 훨씬 중요하다고 생각하는 '자유'라는 이념을 피력하기 위해서 '자유 소프트웨어'라는 용어를 계속해서 사용할 것이다.

노력하라!

요다의 철학(There is no 'try')[14]은 멋지게 들리지만 내게는 효과가 없는 말이었다. 나는 내가 이 일을 해낼 수 있을까 하는 걱정과 우려 속에서 대부분의 일을 해왔다. 그러나 내 도시를 침략하려는 적과 내 도시 사이에는 오직 나밖에 없었기 때

문에 나는 최선을 다해서 싸웠으며 때때로 나 자신이 놀랄 정도로 승리를 거두곤 했다. 때로는 패배하기도 해서 나의 도시 중 몇몇이 파멸되기도 했다. 그러나 또 다른 도시가 위협받음을 발견했을 때는 비장하게 또 다른 전투를 준비해왔다. 싸움이 길어지면서 나는 전운을 감지하는 방법을 체득하게 되었고 다른 해커의 동맹과 원조를 요청하면서 방어할 도시로 떠나곤 했다.

그러나 나는 이제 혼자가 아니다. 전선을 방어하기 위해서 사투를 벌이는 해커 부대를 볼 때 나는 큰 기쁨과 위안을 느낀다. 아마도 이 도시는 건재하리라. 그러나 최근 몇 년 사이에 위협은 더욱 커져가고 있으며 마이크로소프트는 우리의 공동체를 노골적으로 위협하고 있다. 우리는 우리가 가진 자유의 미래가 보장되리라고 확신할 수 없다. 자유가 남아 있게 되리라는 막연한 생각은 버려야 한다! 만약 여러분의 자유가 계속해서 지켜지기를 원한다면 자유를 지킬 준비를 해야만 할 것이다.

---

01 역자주_해커라는 단어가 시스템의 보안을 뚫고 들어가서 해악을 끼치는 일을 저지르는 침범자의 의미로 사용되고 있는 현재의 관행은 대중 매체에 의해서 오도된 경향이 크다고 할 수 있다. 우리 해커들은 이 단어를 침범자라는 의미가 아닌 "프로그램을 만들기 좋아하고 능숙하게 다룰 수 있는, 즉 프로그램 자체를 즐기는 사람"이라는 의미로 사용하고 있다.

02 역자주_스핀오프(spin-off) 회사란 대학이나 기업 연구실의 연구원들이 독립해서 설립하거나 깊이 관계하는 회사를 의미한다.

03 역자주_원제는 『Hackers: Heroes of the Computer Revolution』이며, 우리나라에서는 사민서각에 의해서 『해커-그 광기와 비밀의 기록』이라는 제목으로 번역 출간되었다. 특히 이 책의 에필로그인 '진정한 해커의 종말-케임브리지 1983년' 부분에는 리처드 스톨만과 MIT 인공지능 연구소를 배경으로 한 자세한 뒷이야기들이 수록되어 있다.

04 무신론자로서 나는 어떠한 종교 지도자도 따르지 않지만, 때때로 그들이 남긴 금언에 내가 탄복하고 있다는 사실을 깨닫곤 한다(역자주_힐렐은 예수 탄생 전에 실존했던 유대 랍비의 이름이며, 인용된 힐렐의 금언은

탈무드 토라의 '위대한 세 명의 랍비' 장에서 인용된 것이다).

05 역자주_우리말의 경우에는 한자 문화권의 영향으로 이러한 두 가지 경우에 대해서 무료(無料)나 무상(無償)이라는 단어와 자유(自由)라는 단어를 선택적으로 사용할 수 있다. 따라서 '자유 소프트웨어'라는 용어상에 존재하는 혼란이 영어보다 비교적 적은 편이라고 할 수 있다. 그러나 이러한 언어상의 차이에 관계없이 중요한 것은 '자유'가 의미하는 본질적인 부분이다.

06 역자주_이러한 점들은 영어뿐만 아니라 한국어에도 동일하게 적용되는 것으로 자유 소프트웨어 대신에 무료나 공개, 해방 소프트웨어와 같은 단어는 원어인 free가 가진 의미를 그대로 살리지 못한다고 볼 수 있다. 따라서 '자유 소프트웨어'라는 용어가 가장 좋은 형태라고 생각한다.

07 역자주_리처드 스톨만에게 도착한 회신은 독일어가 아닌 영어로 되어 있었기 때문에 free라는 영어 단어가 정확히 어떤 의미로 사용되었는지 자신도 알 수 없다는 리처드 스톨만의 확인이 있었다. 단지 정확하게 말할 수 있는 것은 VUCK가 '자유 소프트웨어'는 아니었다는 점이다. 아마도 V는 Verfuegung이나 Verfuegungsfreiheit의 첫 글자로 추측해볼 수 있지만 이를 확인할 방법은 없다.

08 1984년인지 1985년인지 정확히 기억나지는 않지만 돈 홉킨스(Don Hopkins)라는 사람이 내게 편지를 보내온 적이 있었다(그는 상상력이 매우 풍부한 친구였다). 그가 보낸 편지봉투에는 재미있는 문구가 몇 개 쓰여 있었는데 그중에 "카피레프트 -모든 권리는 취소됩니다."라는 구절이 있었고 나는 '카피레프트'라는 단어를 당시에 내가 생각하고 있던 배포 형태를 나타내는 이름으로 사용하게 되었다.

09 역자주_GNU(General Public License)는 소프트웨어를 배포하기 위해서 GNU가 사용하는 일반적이고 공개적인 라이선스라는 의미를 가졌기 때문에 'GNU 일반 공개 라이선스'쯤으로 번역할 수 있지만, 대부분의 경우에는 GPL이라는 용어를 사용하므로 우리말로 번역하지 않고 GPL 또는 원어 그대로 GNU(General Public License)라고 하는 것이 가장 바람직한 표현 형태다.

10 역자주_'Bourneagain Shell'이라는 이름은 유닉스에서 일반적으로 사용되던 셸 프로그램인 'BourneShell(본셸)'에 빗대어 만든 재미있는 표현이다. BASH는 다른 셸들과는 달리 약어 그대로 배시라고 발음하며, Bourneagain은 발음상 Bornagain이 되어 최초의 셸인 본셸의 재현이라는 의미를 갖는다.

11 역자주_에릭 레이먼드의 대표적인 저작인 『성당과 시장』에 소개된 문장으로 http://gnu.kldp.org/cathedral-bazaar 사이트를 통해서 한국어 번역 전문을 참고할 수 있다.

12 역자주_GNOME의 약자를 구성하는 단어들은 여러 가지 형태로 배열할 수 있는데 의미의 전달이 가장 명확한 형태는 GNU's Environment Model for Network Objects다.

13 역자주_GNOME 프로젝트의 리더인 미구엘 드이카자(Miguel delcaza)는 1999년에 미국 의과학 전문지

『MIT Technology Review』가 창간 100주년 특집호에서 선정한 '다음 세기를 이끌 젊은 과학자 100인'에 선정되기도 했으며, 1999년 12월에는 자유 소프트웨어 재단이 선정한 '제2회 자유 소프트웨어 발전을 위한 자유 소프트웨어 재단상'을 수상하기도 했다. 자유 소프트웨어 재단은 자유 소프트웨어의 발전을 위해서 지대한 공헌을 한 사람을 매년 선정해서 자유 소프트웨어 재단상을 수여하는데, 1회 때는 Perl의 개발자인 래리월(Larry Wall)이 이 상을 받았다.

14 역자주_There is no "try"는 영화 스타워즈 에피소드 5에 나오는 제다이 스승 요다의 말로 "시도해본다는 것은 의미가 없고 확신을 하고 실행하는 것만이 포스의 힘을 불러낼 수 있다"는 의미다. 원문의 내용은 다음과 같다. "No! Try not. Do. Or do not.There is no try."

5 | 시그너스 솔루션의 미래

마이클 티만Michael Tiemann

송창훈 역

1989년에 설립된 시그너스 솔루션[01]은 오픈 소스에 기반한 최초의 기업이었으며, 1998년 8월에 조사된 포브스지의 자료[02]에 따르면 현재까지도 가장 규모가 큰 오픈 소스 기업이다. 시그너스 솔루션의 주력 상품인 GNU Pro는 임베디드 소프트웨어 개발 도구 시장에서 가장 각광받는 컴파일러와 디버거라고 할 수 있는데, 세계 최고의 마이크로프로세서 제작 회사를 비롯해서 가전 제품과 인터넷, 텔레커뮤니케이션, 사무 자동화, 네트워킹, 항공, 우주, 그리고 자동차 업체가 모두 우리의 고객층이다.

캘리포니아 서니베일Sunnyvale에 위치한 본사를 비롯해서 미국 애틀랜타와 보스턴, 그리고 영국 캠브리지, 일본 도쿄, 캐나다의 토론토에 지사를 두고 있으며 오스트레일리아에서 오레곤에 이르는 많은 지역에 임직원을 둔 시그너스 솔루션는 임베디드 소프트웨어 부문에서 제일 큰 비공개 회사privately-held company이며, 공개 회사publicly-held company 부문에서도 이미 2개의 회사를 추월해서 세 번째 규모의 공개 회사를 앞지르는 시점에 있다. 또한 시그너스 솔루션은 1992년 이래로 CAGRCompound Annual Growth Rate, 복합 연평균 성장률이 65%를 넘으며, 산호세 비즈니스 저널[03]이 선정하는 최고 성장 비공개 회사 100위 안에도 3년 동안 포함되었고 이제는 전 세계 소프트웨어 업체의 총 소득액 기준으로 선정되는 소프트웨어 500 리스트[04]에도 포함되어 있다.

나는 이 글을 통해서 우리에게 성공의 청사진을 제공했던 오픈 소스의 모델과 앞으로 개발될 우리의 제품을 위해 오픈 소스 모델을 어떻게 수정하고 발전시켜 나갈 것인가에 대해서 설명하려고 한다.

1989년 11월 13일, 우리는 캘리포니아 기업부[05]로부터 그토록 기다리던 사업 승인서를 받았고 6천 달러의 금액을 예치한 뒤에 '시그너스 서포트Cygnus Support'라는 이름으로 사업을 시작하게 되었다. 그날은 2년 전부터 준비해왔던 우리의 소망이 실현된 날이었으며, 10년이 넘는 오늘날까지도 계속되는 사업의 긴 여정이 시작된 날이었다.

미래의 사업 전망은 무척이나 순수하게 시작된 것이었다. 한 번은 아버지께서 이런 말씀을 하신 적이 있었다. '얘야, 책을 읽을 때는 처음부터 끝까지 모두 읽도록 하렴.' 아버지들이 당부하는 대부분의 충고와 마찬가지로 나는 마음이 내킬 경우에만 그 말을 따랐는데, 1987년에 하던 일에 싫증을 느끼고 GNU 소프트웨어에 관심을 갖게 되었을 때 나는 리처드 스톨만이 직접 출판했던 GNU Emacs 매뉴얼[06]을 처음부터 끝까지 모두 읽을 결심을 하게 되었다(리처드 스톨만이 책을 직접 출판했던 이유는 사용자가 출판물을 자유롭게 복제해서 재배포할 수 있도록 허용하려는 그의 생각을 받아들인 출판사가 없었기 때문인데, 사실 오늘날에도 이러한 형태의 출판 방식을 도입할 수 있는 출판사는 많지 않다).

Emacs는 정말 환상적인 프로그램이었다. 텍스트 에디터의 기능은 물론이고 사용자의 취향에 맞게 사용자화customize 과정을 거치면, 메일을 처리하거나 뉴스 그룹에 글을 올릴 수 있었으며, 셸을 실행하거나 프로그램을 직접 컴파일하고 디버깅할 수 있었다. 심지어 프로그램을 구동하는 LISP 인터프리터 자체에 대한 조작도 대화형으로 접근이 가능했다. 창의적인 사용자나 일상적인 것에 실증을 느낀 해커들은 Emacs에 별의별 기능을 추가했다. 예를 들어 조셉 와이젠바움Joseph Weizenbaum이 개발한 ELIZA 프로그램의 형태와 같은 대화형 심리 치료 과정이 진행되는

'doctor 모드'[07]라든가 문서의 내용을 뒤죽박죽으로 만들어서 글을 재미있게 읽어 볼 수 있는 'dissociated-press 모드'[08], 그리고 심지어는 하노이 탑을 애니메이션화할 수 있는 '하노이 모드'[09] 등이 그러한 것이었다. 바로 이러한 깊이와 풍부함이 나로 하여금 GNU Emacs의 매뉴얼과 소스 코드를 깊이 있게 알고 싶다는 욕심이 들게 만들었던 것이다.

매뉴얼의 마지막 장에는 'GNU 선언문'이라는 글이 포함되어 있었는데 나는 그 글을 통해서 Emacs를 사용하면서 느끼게 된 "왜 이렇게 좋은 프로그램을 자유롭게 배포할 수 있도록 만들었을까?"라는 풀리지 않는 의문에 대한 해답을 얻을 수 있었다. 리처드 스톨만은 이러한 평범한 질문에 대해서 다음과 같이 대답한다.

> 왜 GNU를 작성해야만 했는가?
>
> 어떤 프로그램을 좋아한다면 당연히 그것을 좋아하는 사람들과 함께 나누는 것이 황금률(대우받고자 하는 대로 대하라-성서)이라고 생각한다. 소프트웨어를 판매하는 사람들은 사용자를 각각 구분하고 그들 위에 군림하고, 사용자 서로가 프로그램을 공유하는 것을 막고자 한다. 나는 이런 식으로 사용자 사이의 결속이 깨지는 것을 거부한다.

위 문구 이외에도 GNU 선언문에는 너무도 많은 담론이 들어 있지만 이곳에 모두 인용할 수는 없으므로, http://www.GNU.org/GNU/manifesto.html을 참고하기 바란다. GNU 선언문의 내용은 사회주의자의 항변처럼 느껴질 수도 있다. 그러나 나는 그 안에 교묘하게 위장되어 있는 사업적 계획을 발견할 수 있었다. 주된 생각은 간단하다. 오픈 소스는 전 세계 프로그래머의 성과를 하나로 응집할 수 있다. 그리고 그러한 프로그램에 대한 사용자화나 성능 향상, 버그 수정, 사후 지원 등의 상업 서비스를 제공하는 기업은 오픈 소스라는 새로운 형태의 소프트웨어에 대한 폭넓은 지지 기반과 규모의 경제를 바탕으로 자본화할 수 있는 것이다. 즉, 단위 생산 요소의 투입량을 증가시킴으로써 이익을 증대시키는 규모의 경제를 오픈 소스라는 메커니즘을 통해서 구현함으로써 이윤을 창출할 수 있는 것이다.

자유 소프트웨어 재단의 프로그램 중에서 마음을 흔들만큼 매력적인 것은 비단 Emacs 뿐만은 아니었다. GNU 디버거GDB는 DEC(현재는 컴팩으로 합병되었다)와 썬Sun에서 제공하던 디버거들이 Emacs와 같이 복잡한 프로그램을 디버깅하기에는 역부족이었기 때문에 리처드 스톨만이 직접 만든 프로그램이었다. GDB는 프로그래머들이 필요로 하는 다양한 명령어와 확장 기능을 가져 큰 규모의 작업도 간단하고 산뜻하게 처리할 수 있었다. 또한 GDB는 오픈 소스 소프트웨어였기 때문에 프로그래머들이 GDB에 필요한 기능을 직접 추가해서 좀 더 강력한 프로그램으로 만들 수 있었다. 바로 이러한 점들이 독점 소프트웨어에는 존재하지 않는 '규모의 확장'이었던 것이다.

하지만 좀 더 극적인 사건은 1987년 6월에 리처드 스톨만이 GNU C 컴파일러의 1.0 버전을 발표한 것이었다. 나는 즉시 GCC를 다운받은 뒤에 GNU Emacs와 GDB 매뉴얼에서 읽었던 모든 기법을 동원해서 11만 라인에 달하는 코드를 모두 습득해 버렸다. 스톨만이 발표한 첫 번째 버전의 GCC는 구형 VAX와 신형 Sun 3 워크스테이션 2개의 플랫폼 밖에 지원하지 못했지만, 두 회사가 제공하는 컴파일러보다 뛰어난 성능을 낼 수 있는 코드로 만들어진 것이었다. 나는 2주 동안 GCC를 내셔널 세미컨덕터사National Semiconductor의 32032 마이크로프로세서에 이식하기 시작했는데, 실험 결과 내셔널의 독점 컴파일러보다 20%나 빠른 성능을 보여주었다. 나는 다시 2주간의 작업을 거치면서 성능 차이를 40%까지 끌어올릴 수 있었다. 내셔널의 칩이 사라지게 된 이유는 모토로라의 68020과 경쟁하기 위해 필요했던 1 MIPS[10]에도 못미치는 0.75 MIPS의 속도 때문이었다고 흔히 이야기한다. 그러나 32032가 출시되었던 초기의 애플리케이션 벤치마크 결과가 0.75 MIPS 였더라도 GCC의 속도 향상을 통해 산출할 수 있는 수치는 모토로라와 경쟁 가능한 140 × 0.75 = 1.05 MIPS다. 그렇다면 형편 없는 컴파일러 때문에 내셔널은 얼마나 많은 비용을 허비해야 했을까? 컴파일러와 디버거 그리고 에디터는 프로그래머가 매일 사용하는 세 가지 주요 개발 도구다. 그리고 GCC와 GDB, Emacs는 독점

소프트웨어보다 월등한 성능을 갖고 있었다. 따라서 나는 독점 기술을 그보다 좋은 기술로, 더군다나 계속해서 향상되는 기술로 대체할 수 있다면, 얼마나 많은 경제적 이익이 있을까라는 생각을 하지 않을 수 없었다.

다시 한번 GNU 선언문의 내용을 인용해보면,

> 유해한 수단을 사용하지 않는다면 노동에 대한 보수와 자신의 소득이 극대화되기를 바라는 것에는 아무런 문제가 없다. 그러나 지금까지 소프트웨어 산업에서 보편화된 수단은 유해한 방법이다.
>
> 프로그램을 사용하는 것에 제한을 두어 돈을 벌어들이는 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. 그까닭은 만일 모든 사람이 그렇게 한다면 상호 간의 유해한 행위로 인해 결과적으로 우리 모두가 더욱 빈곤해질 것이기 때문이다.

무거운 주제를 다루고 있지만 종국적으로 GNU 선언문은 이성적인 문서다. GNU 선언문은 소프트웨어와 프로그래밍의 특성과 학문적 교육의 위대한 전통에 대해 자세하게 분석하며, 결국 금전적인 측면에 관계 없이 자신이 공유할 수 있었던 정보는 다른 사람과도 함께 공유해야 한다는 윤리적, 도덕적 의무에 대한 결론을 내리고 있다. 그러나 나는 다른 결론을 갖고 있었다. 이러한 견해 차이에 대해 스톨만과 나는 가끔씩 논쟁을 벌여 왔는데, 나는 소프트웨어를 자유롭게 사용하거나 개작, 배포하려는 자유는 그러한 자유를 제한하는 억압체가 있을 경우에 더욱 극대화된다는 것과 그러한 자유를 증대시키는 힘은 도덕적 미덕이 아닌 시장과 경쟁의 원리에 있다고 생각했다.

나는 공유의 자유를 통한 혁신이 생산 가격을 낮출 수 있는 방법과 공개 표준과 같은 요인을 통해 규모의 경제를 높여갈 수 있는 방법을 설명했는데, 대부분의 사람은 나의 주장에 대해서 다음처럼 대답했다.

"그것은 좋은 생각이다. 그러나 실현되지는 못할 것이다. 왜냐하면 자유 소프트웨어를 구입하기 위해 돈을 지불할 사람은 아무도 없기 때문이다." 그후 2년 동안 나는 좀 더 다듬어진 논리와 생각을 많은 사람에게 이야기 했지만 "그것은 좋은 생각이다. 그러나..." 라는 위치에 여전히 머물러 있었다. 그러던 어느 날, 나는 다음과 같은 두 번째 통찰을 얻게 되었다."모든 사람이 좋은 생각이라고 동의하지만 아무도 그것이 현실화되지 못하리라고 생각하고 있다면... 그렇다. 나는 어떠한 경쟁자도 갖지 않은 것이다!"

$$-F = -M\,A$$

— 아이작 뉴턴[11]

뉴턴의 법칙을 위와 같이 기술하는 물리학 책을 본 사람은 없을 것이다. 그러나 수학적으로 볼 때 이것은 'F = ma'와 동일한 의미를 나타낸다. 이러한 접근 방식의 주안점은 가정을 반대로 적용하는 것이다. 주어진 가정에 대한 대우 명제를 만들면 등식이 이상해 보일 수 있겠지만 원래의 방정식과 같은 결과가 된다. 나는 오픈 소스 소프트웨어에 대한 상업 지원을 제공하는 사업이 불가능한 것처럼 보이는 것은 사람들이 마이너스 기호에 너무 집착한 나머지 양변의 부호를 상쇄해 버릴 생각을 하지 못하기 때문이라고 생각했다.

적의 침략에는 저항할 수 있지만 시대의 대세에는 저항할 수 없다.

— 빅토르 위고[12]

상당한 가정이 개입되기는 했지만 내가 스탠퍼드 대학의 PH. D. 과정을 그만두고 사업을 시작하기 위해서는 해결해야 할 마지막 의문이 하나 남아 있었다. 내가 창업할 분야의 독점 기술을 매수할 수 있는 충분한 자본이 내게 있다고 가정한 상태에서 나는 썬Sun의 기술을 생각했고 디지털 기술을 생각했고, 그리고 내가 알던 모

든 기술을 생각해 보았다. 다른 사람이 GNU에 기반한 오픈 소스 기업을 만들어 내 사업을 잠식하고 추월하기 전까지 과연 나는 사업에 성공할 수 있을 것인가? 초기 자본을 회수할 수 있을까? 내가 오픈 소스 업체와 경쟁할 경우를 가정한 마지막 시나리오를 검토해 보고 오픈 소스 소프트웨어와 경쟁하는 것이 얼마나 불리한 위치에 서는 것인지를 깨달았을 때, 나는 오픈 소스 소프트웨어가 이미 시대의 대세라는 사실을 절감하게 되었다.

이론과 실제의 차이가 이론에서는 매우 작은 듯이 보이지만 실제는 매우 큰 것이다.

— 작가 미상

이제 오픈 소스 사업 모델을 뒷받침하는 구체적인 이론과 우리가 이러한 이론을 실제로 적용한 방법을 설명해 보고자 한다. 먼저 몇 개의 유명한 문구로부터 시작해 보기로 하자.

자유 시장은 최대 가치를 창출하려고 자원을 가장 효율적으로 사용하며 스스로 조직된다.

— 애덤 스미스[13]

정보는 만들어지는 데 사용된 비용에 상관 없이 무료나 그에 준하는 비용으로 복제되고 공유될 수 있어야 한다.

— 토마스 제퍼슨[14]

자유 시장 경제의 개념은 너무나 방대한 것이어서 나는 매년 노벨 경제학상 수상 시기가 돌아올 때면 애덤 스미스를 가장 유창하고 쉽게 설명한 사람에게 상이 돌아갈 것이라고 농담삼아 말하곤 한다. 그러나 이러한 농담 뒤에는 핵심 사실이 있다. 소프트웨어에 좀 더 적합한 자유 시장 체제는 아직까지 발현되지 않던 무한한 경제적 잠재력을 일깨워 낼 수 있다는 점이다.

애덤 스미스 시절의 자유 시장 경제는 개인이 직접 여행하거나 무역할 수 있는 제한된 범위에서만 가능했을 뿐이며, 국가 사이의 거래와 같은 좀 더 큰 규모의 무역

에는 많은 통제가 뒤따랐다. 따라서 많은 수의 상인이 기존의 인세 기반 무역 제도에 반발하게 되었을 때, 그들은 봉기를 일으켰고 이전의 어떠한 정부보다도 그들의 활동에 자유를 줄 수 있는 새로운 정부를 만들어 내었던 것이다. 실제로 이러한 자유는 미합중국 헌법의 근간을 이루는 핵심적인 사상과 이상이며, 오늘날 국제 경제와 정치에 있어 모든 중요한 협력과 활동의 중심이기도 하다. 그렇다면 무엇이 자유를 이렇게도 중요한 관건으로 만들었을까? 자유가 경제적 번영을 책임지는 가장 큰 요인이 된 이유는 무엇일까? 우리는 이 문제에 대해서 간단하게 짚고 넘어가야 될 것 같다.

문제의 오류를 좀 더 깊이 이해할수록 그 문제는 더욱 가치 있는 것이 되어간다.

— 윌리엄 톰슨 켈빈 경[15]

독점 소프트웨어 개발 도구는 대부분 하드웨어 제조업체가 자사의 제품과 함께 공급했는데, 1989년에 프로그래머에게 선보인 개발 도구는 매우 조악한 것이었다. 첫째로 기본적인 기능만 제공되는 단순한 것이었다. 둘째, 사용할 수 있는 기능도 대부분 도구 안에 내장되어 구현된 빌트 인built-in의 한계로 인해서 프로젝트가 복잡해지면 사용할 수 없게 되는 경우가 많았다. 셋째, 독점 업체의 지원이 너무나도 열악했다. 따라서 대량의 하드웨어를 계속해서 구입하는 상황이거나 소프트웨어에 대한 대규모 사이트 라이선스를 갱신하는 방법을 통해 약점을 보완할 수 있는 충분한 재정이 확보되어 있지 않은 상황에서는 빌트 인 기능으로 인한 곤란을 겪기가 마련이었다.

마지막으로 모든 하드웨어 업체가 자신만의 고유한 독점 사양에 맞춰서 제품을 구현했기 때문에, 만약 특정한 기능이 빈약한 플랫폼용으로 제공된 개발 도구로 프로그램을 만들게 되면 처음에는 별다른 문제가 없을지 몰라도 점점 더 문제가 커져서 결국 해당 플랫폼에 종속되어 버리는 상황에 빠지게 되었다. 이러한 모든 점으로부터 자명하게 알 수 있듯이 자유 시장 경제의 미덕이 무엇이었든 간에 소프트웨어

시장에서는 그것이 유효하지 않았다. 이러한 사례는 분명히 잘못된 것이었으며, 독점 소프트웨어 모델이 갖는 오류에 대한 매우 가치 있는 교훈을 남겨 주었다.

오늘날에도 자유 시장 경제는 독점 소프트웨어 회사의 장벽 안에 갇혀 있다. 그 안에서는 엔지니어와 경쟁 그룹이 연구비와 지원을 얻기 위해서 경쟁하는 자유 시장의 모델이 유효하다. 그러나 그들의 독점 장벽 밖에서는 라이선스 협약과 특허 그리고 거래 상의 비밀로 인해서 소프트웨어의 사용과 배포가 매우 심하게 통제되고 있으며, 사람들은 거시적이 아닌 미시적인 차원에서, 즉 개인에게 소프트웨어의 사용과 배포에 대한 자유가 허용된다면 독점 소프트웨어 회사에 의해서 공급되던 프로그램에 존재하던 힘과 효율성이 없어지지는 않을까라는 우려와 추측을 해볼 수 있을 뿐이다. 독점 소프트웨어 회사에 의해서 자유가 배제되어 버린 독점 시장 경제의 문제에 대해 우리는 사용자에게 소스 코드 수준의 지원을 제공하는 기업을 시작하는 것으로 해답을 찾아 나가려고 했다.

발명은 1%의 영감과 99%의 노력으로 이루어진다.

— 토마스 에디슨[16]

소프트웨어 회사가 가진 가장 간단한 시각은 다음과 같은 것이다. 만약 사람들이 구입하기를 원하는 소프트웨어를 만들었다면 해당 소프트웨어를 복제하고 배포하는 행위는 화폐를 인쇄하는 것과 다르지 않다. 상품을 만드는 데 소요된 비용은 매우 미미한 것이며, 따라서 그 이윤은 매우 크다. 나는 1980년대에 소프트웨어와 관련된 제반 상황들이 열악한 상태를 벗어나지 못했던 이유의 하나는 사람들이 소프트웨어를 널리 사용하게 되면 소프트웨어의 사용과 배포에 대한 어떠한 형태의 풍토가 새롭게 만들어지리라는 점을 인식하지 않고, 단지 돈을 찍어내는 것과 같은 추상적인 모델에 지나치게 집착했기 때문이라고 생각한다.

또한 소프트웨어 회사들은 소프트웨어에 대한 지원을 제공하는 개념이 마치 소프트웨어 제조 공정 상의 결함으로 생겨난 부산물을 제거하는 것처럼 보일 수 있고,

무엇보다도 사후 지원에대한 투자를 줄이면 이윤을 극대화시킬 수 있다고 판단하는 것이다.

쉽게 구현될 수 있는 기능은 "전략적이지 못하다."는 이유로 흔히 사라지곤 한다. 그렇지 않은 경우에는 소스 코드에 대한 접근 없이는 사용자가 해결할 수 있는 부분도 추측과 분쟁의 여지로 남아 있을 수밖에 없게 된다. 또한 궁극적으로 고객이 아닌 제조 업체와 그들의 마케팅 담당 부서는 쉽게 구현될 수 있는 기능임에도 불구하고 쓸데 없이 많은 부분을 경쟁을 위한 도구로 규정해 버린다. 이러한 부분은 사용자를 곤혹스럽게 만들 뿐만 아니라 소프트웨어에도 좋지 않은 것이며, 자유 시장 경제를 무너지게 만드는 것이다.

진실은 아무도 독점할 수 없다.

— 작가 미상

보통 법은 모든 사람이 자유롭고 평등하게 이용할 수 있는 법전이다.

— 마이클 티만

자유 시장 경제와 같이 이 세상을 좀 더 살기 좋은 곳으로 만들 수 있는 훌륭한 이론을 갖는다는 것은 무척이나 바람직한 일이다. 그러나 이러한 이론이 스스로 운영될 수 있는 시점이 되기 전까지 필요한 기금을 조성하는 것은 전혀 별개의 문제다. 우리가 구상했던 기업을 만들기 위해 필요한 기금을 조성하려고 했을 때 문제의 소지가 있다고 평가되었던 것은 사용자에게 지원을 제공하는 서비스 기반의 기업이 과연 성공할 수 있겠느냐는 것이었다. 그러나 서비스를 제공하는 방법에 기반을 둔 회사가 소프트웨어 업계에서는 아주 드물더라도 다른 분야에서는 많은 사례를 찾아 볼 수 있다.

미국이나 영국에서 법이 적용되는 예를 한 번 생각해보자. 보통법[17]은 원하는 누구나 자유롭게 이용할 수 있는 법전이다. 어떤 사람도 로에 대 웨이드 사건[18]의 판례

를 변론에 이용하기 위해 별도의 라이선스 비용을 지불할 필요가 없다. 일단 판결이 선고된 뒤에는 소송 비용으로 얼마가 소요되는가와 상관 없이 그 판례는 모든 사람들이 자유롭게 이용할 수 있는 것이다. 그러나 이러한 모든 자유에도 불구하고 변호사는 가장 값비싼 비용을 지불해야 하는 전문가일 것이다. 그렇다면 어떻게 해서 일차적으로 전혀 독점적이지 않은 법전을 이용한 법률적 사업 행위가 그렇게 많은 가치를 갖게 되는 것일까?

사람들이 그렇게 높은 가치를 부여하는 것은 소송 과정 뿐만 아니라 소송이 갖고 있는 가치의 축적에 있다. 만약 여러분이 유능한 변호사를 선임할 수 있다면 소송 과정에서 여러분에게 유리한 판결을 이끌어낼 수 있으며, 그 결과는 새로운 판례가 되어 기존의 법에 새롭게 추가될 것이다. 즉, 변호사의 선임 여부에 따라 승소할 수 있는 가능성이 달라질 뿐만 아니라 승소할 경우에는 자신의 소송 결과가 법의 일부로 포함될 수 있기 때문에 이러한 가치의 축적에 대한 부분이 좀 더 높은 비용을 요구하게 되는 것이다. 그러나 현재의 법은 지금까지 전해오는 수많은 판례로 이루어져 있으므로 능력 있는 변호사라고 해서 마음대로 판결이 좌우되지는 않는다.

보통법을 이용한 법률 사업은 오픈 소스 소프트웨어를 이용해서 표준을 만들고 유지하는 작업과 유사한 개념이라고 할 수 있다. 표준을 만들고 이를 유용한 것으로 만드는 데는 많은 비용이 필요하다. 그러나 표준 없이 작업하거나 실익이 없는 표준을 유지하려고 할 때는 더욱 값비싼 비용이 요구된다.

따라서 미래의 표준이 될 소프트웨어를 유능한 사람에게 맡기는 것은 무척이나 가치 있는 일이다. 우리는 사업 초기에 이러한 가치의 메커니즘을 이해할 수 있다면, 우리에게 소프트웨어 업계에서 사실상의 표준이 될 수 있는 높은 품질의 오픈 소스 프로그램을 개발할 자본을 제공해줄 사람이 있을 것이라고 확신했다.

시그너스 솔루션의 초기 시절

이러한 이론으로 무장한 뒤에 우리의 이론을 실제로 적용해야 할 시점이 다가왔다. 사업에 대해 조금이라도 아는 상태라면 서비스 기반의 사업을 시작하는 것은 별로 어려운 일이 아니다. 그러나 불행하게도 시그너스 솔루션의 공동 창업자 3명[19]은 모두 사업에 대한 경험이 없는 사람이었다.

> 항상 새로운 실수를 만들어낸다.
>
> — 에스더 다이슨[20]

우리는 Nolo 출판사[21]에서 나온 책을 참고해 사업 계획을 수립하고, 정관을 만들고, 그밖의 여러 가지 외적 요건들을 구비해 나갔다. 첫 해에는 가능한 모든 비용을 아끼려고 노력했지만, 결국 그 해에 절약한 돈은 나중에 몇 천배에 해당하는 돈으로 중복해서 지출되었다. 첫 해에 우리가 받은 경영 자문 또한 후에 그 오류를 고치기 위해 수만 배가 넘는 비용을 지불해야 했는데, 그 당시에 받았던 자문도 시간당 수백 달러에 해당하는 것이었기 때문에 초반에 좀 더 유능한 컨설팅을 받을 수 있었을지는 확신할 수 없을 것 같다. 비용을 효율적으로 사용하지 못했던 대부분의 이유는 우리가 접촉했던 변호사가 유능하지 못했기 때문이라기 보다는 그 당시에 우리가 갖고 있던 법률적, 경영적 문제에 대한 판단력의 부족으로 인해 자문을 제공했던 변호사에게 적절한 질문과 요구를 하지 못했기 때문이라고 할 수 있다.

기존의 것과는 다른 전혀 새로운 형태의 사업 모델을 도입했기 때문에 우리는 재정과 회계, 마케팅, 판매, 고객 정보, 그리고 서비스에 대한 새로운 개념을 만들어야 했다. 이러한 작업에 처음 1년이 소요되었지만 모든 것은 혼란스럽기 그지 없었고 모든 사람은 사업을 정상 궤도에 올릴 수 있는 방법을 찾기 위해 애를 태웠다. 그러나 모든 부분은 사업이 진척됨에 따라 완벽하게 재편될 필요가 있었다.

— 시그너스, 우리는 값 싼 자유 소프트웨어를 만든다.

— 존 길모어[22]

이러한 혼란을 없애기 위해 우리는 기본적인 사업 전제를 최대한 단순화시키기 위해 노력했다. 검증된 소프트웨어에 대해 검증된 지원을 제공했으며 규모의 경제를 이용해 이윤을 창출해 내려고 했다. 우리는 소프트웨어의 설치나 기술 컨설팅을 제공하는 지원 서비스와 GCC 컴파일러를 특정한 플랫폼에 이식하는 것과 같은 종류의 개발 서비스를 제공했는데, 우리의 판단을 토대로 이러한 서비스 부문에 내부 인력이 감당할 수 있는 한도의 2배에서 4배의 목표량을 설정했으며 내부 소요 비용의 1/2 내지 1/4의 가격으로 서비스를 판매하려고 했다. 오픈 소스 소프트웨어에 관련된 다른 종류의 사업은 진행시키지 않았는데 그 이유는 두 가지 종류의 서비스 이외에는 아직까지 상품성이 불투명했기 때문이었다. 우리는 고객에게 좋은 개발 도구를 좀 더 저렴한 비용으로 공급하는 데 초점을 맞추었으며 계약이 거듭됨에 따라 사업 수완을 쌓아 나갈 수 있었다.

우리의 첫 번째 계약은 1990년 2월에 이루어졌다. 그리고 4월이 끝나갈 무렵에는 15만 달러 상당의 계약을 성사시킬 수 있었다. 5월에는 우리의 지원 서비스에 관심을 갖는 50군데의 예비 고객에게 홍보 문서를 발송했으며 6월에는 또 다른 100여 군데에 홍보 문서를 보냈다. 어느 날 갑자기 우리의 사업은 현실로 나타났다. 첫 해가 끝나 갈 무렵에 우리는 72만 5천달러 상당의 지원 및 개발 계약을 맺게 되었고 관심을 갖는 모든 분야가 곧 기회의 땅처럼 여겨졌다.

이러한 모든 성공에도 불구하고 우리는 곧 심각한 문제에 봉착하게 되었다. 지원 서비스와 개발 서비스를 내부 소요 비용의 1/2~1/4에 해당하는 가격으로 제공하게 되자 서비스를 공급할 때까지 필요한 총 소요 비용이 150만 달러와 300만 달러를 넘었던 것이다. 그러나 당시에는 공동 창업자 3명을 포함해서 1명의 영업 담당과 시간제로 근무하던 1명의 대학원생까지 모두 5명의 인원이 사업에 필요한 모든

일을 처리해야 했기 때문에 그 인원으로는 확보된 일과 예산에 맞는 마진을 도저히 맞출 수 없었다. "이러한 문제를 해결할 수 있는 전제였던 규모의 경제가 가능하게 될 시점까지 우리의 사업은 과연 얼마나 더 커지게 될까? 현재의 성장률대로라면 규모의 경제가 가능하게 되기 전까지 얼마나 더 많은 사람을 고용해야 할까?" 아무도 이 문제에 대한 해답을 알 수 없었다. 왜냐하면 우리는 선례가 될만한 어떠한 형태의 오픈 소스 사업 운영 모델이나 재정 모델도 갖지 않았기 때문이다.

GNU Pro

우리는 이러한 문제가 심각해지기 전에 규모의 경제를 이루어야 한다는 결론을 내렸다. 그리고 엔지니어답게 규모의 경제를 이루는 가장 빠른 길은 소규모로 특화된 오픈 소스 기술을 이용해서 대량 판매가 가능한 솔루션을 구성하는 방법이라는 결론을 내렸다. 선택할 오픈 소스의 기술과 솔루션의 폭을 적게 잡으면 적게 잡을수록 소품종 대량 생산에 의한 규모의 경제가 좀 더 쉽게 성취될 수 있으리라고 생각했다.

> 최우선적으로 튼튼한 기반을 확보하라.
>
> — 손자[23]

셸과 파일 유틸리티, 소스 코드 제어 소프트웨어, 그리고 심지어는 인텔의 386 아키텍처에 맞는 커널을 개발하려고 했던 최초의 계획을 모두 취소한 뒤에 우리는 GNU 컴파일러와 디버거를 판매하는 것으로 사업 방향을 단일화했다. 우리가 사업을 시작했던 당시에는 썬Sun이나 HP, IBM 등과 같이 자체적으로 컴파일러를 제공하는 10여 개의 기업과 별도의 32비트 컴파일러를 판매하는 10여 개의 서드 파티 업체가 이미 존재했다. 또한 컴파일러를 판매하는 업체가 계속해서 늘어감에 따라서 우리가 32비트 컴파일러 시장을 석권하기 위해서는 사업을 시작할 때 생각했던 기능을 모두 통합시키기 위해 인원 보강 등의 방법으로 사업 규모를 충분히 확

장시켜야만 했는데, 마치 EDS[24]가 IBM 시스템에 대한 아웃소싱을 담당했던 것과 같이 우리에게는 오픈 소스가 규모의 확장을 대신해 줄 수 있었다.

GNU 컴파일러는 이미 10여 개씩의 호스트와 타깃[25] 아키텍처로 이식된 상태였으며, 가장 많은 아키텍처로 이식된 컴파일러의 하나였다(6개의 포팅 작업은 내가 직접 수행한 것이다). GNU 디버거는 5개 정도의 네이티브 플랫폼[26]에서 사용되었으며 몇몇 사람에 의해 임베디드 시스템용으로 이식되기도 했다. 우리는 여러 개의 구성 요소를 취합해서 하나의 단일 배포판으로 만든 뒤에 README 파일을 작성하고, 설치 스크립트를 추가하고, 그 외의 부수적인 프로그램을 통합해서 제품을 출시할 수 있으리라고 생각했지만 실제의 작업은 그렇게 간단한 것이 아니었다.

첫째로 GCC의 버전이 1.44에서 2.0으로 옮겨가는 상태였다. 1.x대 버전의 GCC는 m68k나 VAX와 같은 CISC 머신에서 다른 컴파일러보다 월등한 성능을 가졌지만 RISC 플랫폼에서 경쟁력을 갖추기 위해서는 최적화해야 할 부분이 남아 있었다. 1988년에 내가 GCC를 처음으로 SPARC(스팍) 시스템으로 이식했을 때는 썬 Sun의 컴파일러에 비해서 20% 정도 속도가 뒤쳐져 있었지만 1989년에 인스트럭션 스케줄러를 만들어 그 차이를 10%로 근접하게 만들었으며 같은 해에 브랜치 스케줄러를 추가해 5% 이내로 줄일 수 있었다. CISC에서 RISC로 전환되는 세계적인 추세[27]와 함께 더 이상 기존의 GCC 컴파일러를 그대로 판매할 수 없는 상황이었기 때문에 우리는 고객이 만족하고 선택할 수 있는 새로운 기능과 성능을 구현해야만 했다.

둘째로 GNU C++ 의 개발이 미진한 상태였다. 나는 1987년 가을에 GNU C++를 작성해서 첫 번째 네이티브 C++ 컴파일러를 선보였다. 그러나 C++는 C보다 복잡한 언어였으므로 시그너스를 시작할 때도 여전히 개발을 완료하지 못한 상태였다. 1990년에 새롭고 복잡한 기능이 C++의 표준으로 추가된 데다가 시그너스를 시작하면서 발생한 여러 가지 업무들로 인해 나는 C++를 유지할 만한 시간이 없었다.

셋째로 GDB의 변형 버전이 너무 많이 존재했다. GCC와 G++[28]이 주요 배포 사이트를 통해 정기적으로 발표되는 상당히 일관된 체계를 갖고 있었던데 비해 GDB는 산발적이고 이산적이었다. 오픈 소스의 반대자들은 어떠한 버전도 '표준'으로 인정될 수 없다는 오픈 소스의 문제를 지적하면서 하나의 단일 버전을 사용하는 데 따른 독점 소프트웨어의 장점을 주장한다. 그러나 이는 전 세계에 있는 수많은 사람에 의해 그들의 필요에 맞는 프로그램이 만들어지는 과정에서 버전 상황을 조절해 줄 수 있는 관리자가 없었기 때문이다.

넷째로 우리는 바이너리 관련 작업을 수행할 수 있는 완벽한 소프트웨어 체계를 갖지 못했다. 어셈블러와 링커그리고 binutils라는 이름으로 알려진 바이너리 유틸리티는 GCC와 GDB가 지원되는 플랫폼 중 일부에서만 사용할 수 있었다. 그때까지만 해도 GCC와 GDB가 지원되면서 GNU 어셈블러와 로더인 GAS와 GLD, 그리고 그밖의 바이너리 유틸리티를 모두 지원하는 동일한 소스 코드 기반의 작업이 가능한 플랫폼은 거의 존재하지 않는 상황이었다.

다섯째로 우리는 C 라이브러리를 갖고 있지 않았다. 이것은 썬Sun이나 HP와 같은 네이티브 플랫폼 상에서는 별로 중요한 문제가 아니지만 임베디드 시스템 개발자에게는 매우 큰 문제였다.

여섯째로 우리의 경쟁사는 우리와 대적할 만한 기술을 갖지 않았지만 그들 나름대로 완성된 제품을 이미 효과적으로 판매하는 상태였다. 우리보다 10~100배 정도 매출액이 큰 회사와 경쟁하기 위해 우리는 소규모의 특화된 제품을 만들어 판매하는 방법을 통해 틈새 시장을 이용한 측면 공격이 아닌 전면 공격으로 판매 전략을 수정했다.

그리고 마지막으로 우리 자신의 확신이 문제였다. 빠르게 발전하는 많은 개발 도구를 하나로 통합하는 데 따른 장점은 이러한 도구에 대한 수요가 분명히 존재한다는 것이다. 그러나 우리가 완제품을 내놓았을 때는 제품에 대한 지원 서비스의 수요가

없어질 것이라는 회의론이 제기되었다. 실제로 우리의 사업은 6개월 동안 소강 상태에 접어들었으며 향후 4년 동안 같은 종류의 회의론이 사라지지 않았다.

이 세상은 넘을 수 없는기회들로 가득 차 있다.

— 요기 베라[29]

최초의 계획을 계속해서 추진해 나가는 것밖에 다른 도리가 없었다. 우리는 지난 6개월 동안의 경험을 바탕으로 두 배의 노력을 쏟기로 합의했다. 나는 낮에는 회사의 경영을 책임졌으며 밤에는 GCC 2.0과 G++을 완성하는 작업에 동참했다. 또 다른 공동 창업자인 데이비드 헨켈 월리스David Henkel-Wallace(Gumby라고 부르기도 한다)는 binutils와 라이브러리 작업을 수행했고 지원 업무와 CFOChief Financial Officer, 최고 재정 책임자의 역할을 함께 담당했다. 그리고 존 길모어는 GDB의 작업을 맡았다. 우리는 몇 명의 사원을 새로 고용했는데, 그들은 오픈 소스 버전 관리 시스템인 CVS[30]를 통해 우리의 작업을 관리하고 우리의 제품을 지원 가능한 수백 개의 플랫폼으로 좀 더 쉽게 이식하기 위해 환경 설정과 설치 스크립트를 작성했으며, 제품의 테스트 공정을 자동화하고 그 동안 확장을 자제했던 개발 계약을 체결하는 작업을 도와주었다.

6개월이 지난 후에는 설명할 수 없을 만큼 많은 진척이 있었는데 어떤 사람은 소규모의 특화된 제품을 생산하는 우리의 강도 높은(어떤 사람들은 제한적이라고도 표현했다) 사업 방향에 관심을 잃어갔다. 우리가 개발하거나 판매하는 주력 상품은 GNU 제품이었지만 다른 기술을 활용한 판매 계약을 체결하기도 했는데, 예를 들어 네트워크 보안 소프트웨어인 커버로스Kerberos [31]와 Emacs, 그리고 심지어는 버그 수정과 프레임워크 소프트웨어의 테스트와 같은 일도 포함되었다. 이러한 프로그램은 당시에 모두 개발 단계에 있던 것들이었다.

존은 자신이 새로운 GDB의 관리자가 되겠다고 네트워크를 통해 알렸다. 그리고 만약 다음 버전의 GDB에서 구현되기를 원하는 기능이 있다면 갖고 있는 GDB의

소스 코드를 보내 그들을 통합시킬 수 있게 해달라고 요청했다. 그 후 6주일 동안 존은 137가지 버전의 GDB 소스를 모을 수 있었는데 대부분은 3.5 버전을 그들의 필요에 맞게 수정한 것이었으며 각각의 버전은 통합될 만한 1~2개씩의 기능을 가졌다. 존은 이러한 기능을 모두 지원하기 위해서 GDB 4.0을 디자인했다.

검비는 모든 바이너리 유틸리티가 지금까지 알려진 오브젝트 파일과 디버깅 포맷을 처리할 수 있는 동일한 라이브러리를 사용하겠다고 결정했다. 컴파일러와 함께 연동되는 다양한 종류의 바이너리 유틸리티의 기능을 생각해 본다면 이러한 결정은 분명히 적절한 것이었다.

개발툴	입력 형식	출력 형식[32]
어셈블러	ASCII	ASCII
어셈블러	ASCII	바이너리
Archiver	바이너리	바이너리
링커	바이너리	바이너리
Size	바이너리	바이너리
Strip	바이너리	바이너리
Binary	바이너리	바이너리
Nm	바이너리	바이너리
디버거	바이너리	없음

모든 도구는 바이너리 파일 포맷을 읽고 쓰는 자체적인 구현 방식을 갖고 있었으며, a.out이나 b.out, coff, ecoff, xcoff, elf, IEEE695 등의 포맷[33]을 지원하는데 따른 다른 수준의 구현 방법을 갖고 있었다. 게다가 이러한 도구를 설정할 때는 오직 하나의 바이너리 포맷만을 지원할 수 있도록 고정해 버렸기 때문에 m68k 플랫폼에서 a.out 포맷을 생성할 수 있게 어셈블러를 설정하면 다른 바이너리 유틸리티도 동일한 설정을 해주거나 오브젝트 파일에 독립된 형태로 사용될 수 있게 해주어야 했다. 즉, 유틸리티가 생성하는 파일 포맷에 따라서 하나의 도구를 a.out에 맞게 수정했다면 그와 연관된 모든 도구를 a.out 용으로 변경해야 했던 것이다!

하나의 소스를 기반으로 파생된 기능을 모두 지원할 수 있는 하나의 단일 라이브러리 체계를 구축하면 모든 작업이 일관성 있는 방법으로 구현되거나 관리될 수 있기 때문에 규모의 경제에 좀 더 빨리 도달할 수 있게 된다. 또한 a.out 오브젝트 코드를 coff 라이브러리와 링크시킨 뒤에 IEEE695 실행 파일로 만드는 것과 같은 유연한 조합도 가능해진다. 검비는 이러한 라이브러리의 구조에 대해 리처드 스톨만과 토론하면서 실제 디자인을 만들기 시작했다. 스톨만은 그러한 라이브러리를 구현하기 위해서는 대부분의 도구를 모두 다시 작성해야 하고, 유지하는 것 또한 힘들 기 때문에 무척이나 어려운 작업이 될 것이라고 말했다. 검비 또한 스톨만의 의견에 공감했기 때문에 새로운 라이브러리를 작성하는 일을 '빌어먹을 놈의 빅딜Big Fucking Deal'이라는 말로 표현했는데, 우리는 BFD의 의미를 궁금해하는 고객에게 그것이 Binary File Descriptor Library를 뜻한다고 둘러댔다. 새로운 라이브러리는 이러한 사연 속에서 탄생하게 되었던 것이다.

존과 검비가 열심히 프로그램을 개발할 때, 나는 현금을 동원하기 위해 계약을 유치하려고 바쁘게 움직였다. 매 분기마다 더욱 많은 계약을 체결하기 위해 추가적인 기능이 필요한 제품에 대해 최대한의 목표를 설정했고 유능한 엔지니어는 이러한 제품을 내놓기 위해 전념했다. 그러나 우리가 GNU 소프트웨어에 대한 개발을 많이 하면 할수록 네트워크를 통해 돌아오는 피드백이 적어지는 오픈 소스의 반대 현상이 나타났는데, 이러한 현상은 일련의 GNU 도구에 대한 작업을 50% 수준으로 완성할 때까지 계속되었기 때문에 판매와 개발이라는 두 가지 부문 사이에 긴장을 야기시켰다.

이러한 상황은 일시적인 것이 아니어서 최초의 '진보적인 릴리즈'가 완성되기 위해서 자그마치 1년 6개월이라는 시간이 소모되었다. 제품이 완성되던 역사적인 그날이 되어서야 나는 Sun 3와 Sun 4, 두 개의 플랫폼을 지원할 수 있는 단일 소스 코드 기반의 완벽한 C/C++ 개발 도구가 만들어질 수 있다는 사실을 처음으로 확신하게 되었다. 이보다 짧은 기간 안에 내가 6개 플랫폼에 GCC를 이식하고 3개의

GDB 이식과 1개의 네이티브 C++ 컴파일러와 디버거를 개발했던 것을 생각하면 1년 6개월이라는 기간은 놀랄 만큼 긴 것이었다.

새로운 개발 도구는 기존의 제품에 비해 두 가지 측면에서 개선이 이루어졌다. 첫째, 새롭고 유용한 기능이 많이 추가된 상태에서 이전보다 월등한 성능을 보여주었다. 둘째, 우리가 만든 하부 구조는 도구를 모두 새롭게 작성한 것일 뿐만 아니라 환경 설정 스크립트를 구현하고 프레임워크 테스트를 자동화한 것이었기 때문에 무제한의 가상 임베디드 시스템 지원을 포함해서 좀 더 많은 종류의 호스트/타깃 조합에 대한 지원이 가능해졌고 이러한 프레임워크에 대한 테스트 또한 만족할 만한 결과를 보여주었다.

일시	릴리즈 이름	네이티브	임베디드	총 플랫폼 수
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

엔지니어들이 GNU Pro 툴킷[34]을 개발하기 위해 노력할 때, 판매팀은 이 제품에 대한 적절한 판매 방법을 구상하고 있었다. 1991년에 우리는 소프트웨어 판매 기법을 배우고자 하는 젊은 경영학 전공 학생 한 명을 고용했는데, 당시 어플라이드 매트리얼스Applied Materials사에서 해고된 상태였던 그녀는 영어가 모국어가 아니었음에도 불구하고 업무를 매우 빨리 파악해 나갔다. 그녀는 주말마다 사무실에 남아서 C 언어를 혼자 공부하곤 했으며 해커의 수준은 아니지만 오픈 소스의 활용에 대한 열렬한 지지가가 되었다. 6개월 동안 괄목할 만한 판매 실적을 거둔 뒤에 그녀는 자신이 직접 기안한 고객 프리젠테이션 자료를 보여주었는데, 이는 무척이나 신선

하고 획기적인 것이었다. 나는 해커의 관점에서 항상 오픈 소스의 기술적인 장점을 앞세워 상품을 팔아 왔다. 그러나 그녀는 우리가 수행하는 작업이 가진 본질적인 복잡함과 오픈 소스 소프트웨어의 사업적인 가치에 대해 설명했는데, 이러한 점은 고객에게 자체적으로 소프트웨어를 개발하는 것 대신 우리 제품을 구입하는 것이 좀 더 효율적이라는 것을 설명하는 데 큰 도움이 되었다. 그동안 나는 우리 엔지니어들이 고객이 보유한 인력보다 우수하다는 점을 강조해왔지만, 그들의 인력이 우리보다 열등하다는 사실은 고객의 입장에서 볼 때 결코 유쾌하지 않은 것이다. 그러나 그녀는 고객의 엔지니어가 우리가 이루어 놓은 기본적인 아키텍처 이식과 관련 작업의 성과를 통해 얻을 수 있는 장점을 내세웠으며, 종국에는 우리의 기술 역량과 사업적 장점이 함께 결합되어 고객에게 확실한 수익을 보장해줄 수 있으리라는 점을 설명했다.

연도	매출액($K)	이익률(%)	누적 CAGR(%)[35]
1990년	725	거의 없음	거래 없음
1991년	1500	1	106
1992년	2800	2	96
1993년	4800	3	87
1994년	5700	4	67

큰일 났네, 왓슨군! 어서 이리와 주게!

— 알렉산더 그라함 벨[36]

GNU Pro 툴킷에 쏟은 노력을 바탕으로 만들어진 새롭고 중요한 기술은 오픈 소스의 시작점인 네트워크로 돌아가 또 다른 표준으로서의 역할을 맡게 되었는데, 이러한 프로그램의 예[37]로는 GNU configure 와 autoconf, automake, Deja GNU, GNATS 등이 있다.

GNU Pro는 현재 175개의 호스트/타깃 조합을 지원하는데, 이러한 수치는 시장에서 실제로 판매할 수 있는 하드웨어 조합에 의한 것이지 우리가 보유한 설정 기술이나 배포 기술 상의 한계에 의한 것은 아니다.

사실 GNU Pro의 압도적인 시장 점유율 때문에 몇몇 업체들이 우리와 경쟁하기 위해 GNU 소프트웨어에 대한 상업 지원 의사를 밝힌 적도 있었지만 이 경우에도 오픈 소스 모델은 우리에게 다시 한번 힘이 되어 주었다. 시그너스에서 일하는 엔지니어의 인원 수를 능가하는 많은 수의 인력을 확보하지 않는 한 아무도 '진정한 GNU 소스'라는 우리의 위치를 따라올 수 없다. 왜냐하면 우리의 엔지니어 대부분은 그들 자신이 바로 우리가 지원하는 소프트웨어의 원저작자이거나 주요 관리자이기 때문이다. GCC와 GDB, 그리고 이와 관련된 유틸리티에 대한 수정이 일어난다면 그 중 80% 이상의 작업은 다름 아닌 시그너스의 개발자에 의해 이루어지는 것이다. 이러한 상황에서는 경쟁 업체가 그들의 고객이 원하는 기능을 GNU 소프트웨어에 추가해 판매하는 것 정도는 가능하지만 추가된 기능과 기술 또한 오픈 소스의 메커니즘을 통해 우리에게 다시 돌아오게 되어 있는 것이다. 유용한 기능이라면 시그너스의 기술로 흡수시키면 되고 그렇지 않은 경우에는 그냥 무시하면 된다. 승리 아니면 패배라는 2개의 상반된 측면을 놓고 싸우는 독점 소프트웨어와는 달리 오픈 소스 안에서는 마치 뫼비우스의 띠처럼 모든 개선점은 소프트웨어의 주요 관리자에게 돌아오게 되어 있다. 따라서 우리의 경쟁사가 GNU와 오픈 소스라는 이름을 이용해 마케팅 전략 차원의 이익을 얻을 수 있을지는 모르겠지만, 결국 모든 성과는 시그너스의 이익으로 돌아오는 셈이다. 1989년에 설립해서 오픈 소스 시장을 선도해온 우리의 장점은 다름 아닌 다른 업체와의 경쟁에서 10년 앞서 있다는 점이다.

도전들

CAGR의 비율을 나타낸 표를 살펴보면 시스너스의 성장률이 지속되기는 하지만 조금씩 둔화되고 있다는 점을 알 수 있다. 우리가 오픈 소스 소프트웨어의 가치와 장점을 상품화하려고 노력할 때오픈 소스 모델에 대한 다음 회의론과 잠재 고객들의 의문이 제기되었다.

구매 결정 가능성[38]

경쟁업체에 이득이 될 것을 알면서도 오픈 소스 제품을 구입할 것인가?

성장 가능성

서비스 기반의 사업이 계속해서 성장할 수 있을 것인가?

고객 지원 능력

고객이 만족할 수 있는 서비스를 항상 제공할 수 있을 것인가?

타산성

오픈 소스 소프트웨어로 이윤을 창출할 수 있을 것인가?

유지 및 관리 능력

오픈 소스 소프트웨어가 균일한 품질을 유지하도록 지속적으로 관리할 수 있을 것인가?

투자 가능성

소프트웨어와 관련된 지적 재산권을 갖지 않은 기업에 투자자가 관심을 보일 것인가?

5명의 임베디드 시스템 프로그래머를 둔 관리자에게 1만 달러 상당의 지원 서비스 계약을 판매한다고 할 때, 과연 오픈 소스를 기반으로 성립한 시그너스의 사업 모델이 그대로 유지될 수 있을까? 오픈 소스는 가장 우수하고 혁신적인 소프트웨어 개발 그룹에게 최상의 방법론을 제공함에도 메인스트림 시장에서는 주된 방해물이 된다고 밝혀진 바 있다. 우리에게 있어서 이 시점은 조프리 무어가 그의 책에서 밝힌 '캐즘을 넘어서Crossing the Chasm'[39]라는 말의 의미를 실제로 체득하는 시기였다.

이러한 사실은 내가 포천지가 선정하는 100대 기업[40]중 한 곳에서 무선 통신 시스템을 개발하는 개발자 그룹을 방문했을 때 절실하게 통감할 수 있었다. 품질 관리

공정의 하나로 그들은 상당히 많은 검사 항목에 대해 그들의 제품뿐만 아니라 그들이 구입한 개발 도구에 대한 평가도 함께 수행했는데 다양한 업체의 제품이 모든 검사 항목에 대해서 '우수' 또는 '최우수'의 등급을 받던 데 반해 유독 임베디드 개발 도구만은 품질 관리 공정이 진행된 3년 동안 모든 항목에 대해서 '불량'이나 '불합격' 등급을 받았다. 그러나 기존의 제품이 가진 품질 상의 문제점과 우리 제품에 대한 고객의 호평과 기술 및 가격의 우수성에도 잘 알려지지 않은 솔루션을 도입하지 않으려는 타성 때문에 그들은 우리의 제품을 구입하지 않으려 했다. 나는 실제로 제품을 구입하는 데 참고하지도 않을 것이면서 왜 굳이 제품을 평가하고 관련 자료를 수집하는지 무척 의아해 하면서 돌아왔는데, 나중에 그러한 의문 자체가 잘못된 것이었음을 깨닫게 되었다. 그러한 습성이 메인스트림 시장의 전형적인 행동 방식이라고 받아들이면서 고객의 잘못을 탓하기 전에 우리가 가졌던 기존의 마케팅 방법과 홍보 전략을 개선해야 한다는 사실을 먼저 깨달아야 했던 것이다.

그러나 우리가 가진 문제가 마케팅과 같은 외형적인 곳에만 있는 것은 아니었다. 많은 고객은 우리가 지원 서비스를 확장시켜나가는 데 필요한 인력을 확보할 수 있으리라고 믿지 않았다. 이러한 생각은 옳을 수도 있고 틀릴 수도 있는데, 엔지니어를 고용하는 측면에서 볼 때는 전혀 잘못된 것이라고 할 수 있다. 기본적으로 시그너스는 엔지니어가 만든 회사다. 우리의 문화와 오픈 소스 사업 모델, 그리고 세계 최고의 오픈 소스 개발팀에 합류할 수 있다는 가능성은 그 자체가 이미 개발자 에게 시그너스를 매력적인 곳으로 만드는 요인이 된다. 전국적인, 특히 실리콘밸리의 평균과 비교했을 때 시그너스 개발자의 이직률은 다른 기업의 25%에서 10%선에 불과하다.

그러나 관리직 사원을 고용할 때는 전혀 다른 상황이 되었다. 메인스트림 시장의 고객이 가졌던 생각이나 편견처럼 우리가 접했던 대부분의 사람은 시그너스에서 일하는 것에 별다른 관심을 보이지 않았다. 시그너스에 매력을 느낀 사람은 우리가 생각하는 것과는 다른 이유를 갖고 있었다. 그 당시에 우리는 50명이 넘는 개발자

와 개발 부서를 담당하는 2명의 관리자를 가졌는데 관리자가 오픈 소스를 이해하고 그러한 방식으로 회사를 운영하려고 노력할 때마다 오히려 서로 간의 의사소통과 제품 개발, 경영 관리, 그리고 직원들의 만족도 모두가 감소하는 정반대의 결과를 가져왔다.

이러한 이유 때문에 우리는 아이러니하지만 개발과 경영 방침이 외부에 공개되지 않는 폐쇄적인 요소를 허용하지 않으려는 사람에게는 관리자의 역할을 맡기지 않았다. 오픈 소스는 사업 상의 전략일 뿐이지 철학은 아니라고 생각한다. 따라서 우리는 회사의 목표를 달성하기 위해 오픈 소스와 폐쇄적인 요소를 함께 병행해 관리할 수 있는 사람을 고용하려 했던 것이다.

우리는 오픈 소스 소프트웨어의 합의(含意)를 제대로 이해하는 관리자를 고용할 수 없으리라는 현실을 인정하게 되었다(고용될 관리자가 시행착오를 거칠 수밖에 없다는 사실은 당연하기 때문에 시행착오로 인해서 발생할 수 있는 손실에 대한 예산을 미리 편성해 놓아야 한다). 관리자는 시행착오를 통해 배울 수 있어야 하며 그들의 경험을 통해 시그너스의 맹점을 개선시킬 수 있어야 한다. 그러나 시행착오를 통해서 잘못된 점을 빨리 인식하는 동시에 이를 실제 경영에 즉시 도입할 수 있는, 우리에게 절실히 필요한 그러한 관리자를 발견하는 것은 결코 쉽지 않은 일이었다.

이러한 모든 문제에도 오픈 소스 모델은 탁월한 회복력을 보여주었다. 잘못된 예측이나 집행으로 인해 고객이 감소한 경우도 있었지만 고객이 줄어든 가장 큰 요인은 그들의 내부 결정에 따라서 프로젝트가 철회된 데 따른 것이며, 달러 가치로 산출한 우리의 재계약 갱신률은 1993년부터 연간 90%선을 유지하고 있다. 다른 회사들이 쓰러져 갈 때도 두 가지 요인은 우리가 살아남도록 도와주었는데, 첫 번째는 직위와 연령에 상관 없이 모든 직원이 최우선적으로 고객의 요구를 만족시켜야 한다는 점을 깊이 인식한다는 점이다. 그 누구도 고객 지원보다 우선할 수 없었다. 그리고 두 번째는 우리가 할 수 있는 모든 방법이 실패한 최악의 경우에도 고객이 소

스 코드를 통해 자신의 문제를 스스로 해결할 수 있는 힘을 얻을 수 있었다는 점이다. 이러한 요인을 통해 당시의 시그너스는 내부적으로 놀랄 만큼 많은 혼란이 있었음에도 소프트웨어가 납품되지 않았다는 등의 이유로 거래가 취소되는 일이 거의 없었는데 이러한 사례는 지원 서비스가 제공되지 않는 오픈 소스를 사용하는 사람뿐만 아니라 독점 소프트웨어 업체에서도 찾아보기 힘든 놀라운 것이었다.

오픈 소스를 넘어서 – eCos

임베디드 시스템 업계에서는 소수의 주요 업체만이 임베디드 시스템을 구현할 수 있는 칩을 생산하며, 또한 소수의 주요 임베디드 OEM 업체[41]가 생산한 칩의 대부분을 소비하는 것이 현실이다. 주요 OEM 업체를 제외한 대다수의 소규모 업체가 나름대로 관심 분야의 제품을 만들지만 소량 생산의 특성 상 그들의 제품에 맞는 새로운 칩의 디자인이나 소프트웨어 솔루션이 개발될 수 있는 수요를 충족시키지는 못하고 있다.

반도체 제조업체와 OEM 업체 사이에는 수백 개에 달하는 소규모 소프트웨어 업체가 있으며, 그들은 모두 자체적인 소프트웨어를 판매한다. 그러나 IDC[42]의 자료에 의하면 120개가 넘는 상용 RTOS[43]가 시장에서 판매되지만 이중에서 어느 업체도 6% 이상의 시장 점유율을 갖지 않는다고 한다. 이러한 현상은 10여년 전의 유닉스 시장과 다르지 않으며 단지 분산의 폭이 20배 정도 넓어져 있을 뿐이다. 시장이 분산되는 이러한 현상은 중복과 비호환성 그리고 가격 폭리와 같은 자유 시장 경제를 퇴보시키는 전형적인 요인이 될 수 있다. 반도체 제조업체와 OEM 업체가 원하는 것은 시간과 비용을 절감할 수 있는 표준이며, 상용 RTOS 업체는 시간이나 비용 또는 이 두 가지 모두를 지나치게 소모했던 것이다.

임베디드 시스템 시장에서 우리는 떠오르는 별과 같았다. 우리는 이 시장의 선두 업체보다 2배 이상 빠르게 성장했으며 상위 4개 경쟁업체에 대한 한자리수 성장률을 유지했다. 그러나 우리는 진정한 업계의 1인자로 대우받지 못했으며 또한 그렇

게 행동하지도 않았다. 1995년에 우리는 주요 고객과 함께 임베디스 시스템 업계에서 수행 가능한 일에 대한 많은 대화를 나누면서 기존의 GNU Pro 툴킷이 단지 문제를 짚어내는 역할만 할 수 있다는 사실을 알게 되었다. 고객이 필요로 하는 것은 단순한 문제의 지적이 아니라 임베디드 시스템 개발을 위해 표준 C 라이브러리나 실시간 POSIX API를 직접 사용할 수 있는, 아키텍처와 독립적인 소프트웨어의 추상적 계층 모델이었던 것이다. 이는 우리에게 있어 결코 작지 않은 시장을 새롭게 창출할 수 있다는 가능성을 의미했다.

우리는 RTOS 업계의 현황을 하나둘씩 파악해 나갔다. 120개가 넘는 RTOS 업체의 수와 그들이 내부적으로 사용하는 RTOS의 수가 1000여개가 넘는다는 사실은 지금까지 그 누구도 기술적인 측면에서 '하나로 모든 요건을 충족할 수 있는' 다양한 환경 설정이 가능한 RTOS를 만들지 못했다는 것을 의미한다. 우리는 런타임 run-time 로열티의 마진이 매우 높다는 사실에 주목했다. 새로운 RTOS는 무상으로 제공되어야 하는 것이다.

즉, 주변 시장을 모두 병합하려면 기존의 것과는 다른 새로운 신기술이 개발되어야 하고 그것을 무상으로 공개할 필요가 있었다. 그러나 이러한 구상이 경영적으로 뒷받침되기 위해서는 1년 간의 시간이 필요했다.

일단 이러한 전략을 실행하기로 결정한 후에 경영 관리팀은 "이 제품으로 어떻게 수익을 낼 수 있을 것인가?"라는 문제를 놓고 고민하기 시작했다. GNU Pro를 이용해서 시장을 장악해 나가기는 했지만, 이러한 모델이 임베디드 시장에서도 동일하게 적용될 수 있을지는 분명하지 않았다.

우리는 어떠한 사업이 문제에 봉착했을 때 현재의 상황에 대한 여러 가지 가정을 세워보는 현명한 방법을 이용해 보았다. 즉 '새로운 제품을 이용해서 수익을 올릴 수 있는 방법이 해결되었다고 전제할 때, 고객의 문제를 해결하고 업계 1위의 위치를 차지하기 위해서 처리해야 할 다음 번 과제는 무엇일까?"라는 질문에 대한 해결

방법을 찾아보았던 것이다. 첫째, 다양한 개발 작업을 지원할 수 있는 새로운 설정 기술을 개발해야 한다. 둘째, 고객이 직접 설정 작업을 수행할 수 있도록 관련 도구을 개발해야 한다. 셋째, 이러한 요건을 시장의 수요가 사라지기 전에 모두 완료해야 한다. 소프트웨어 개발에는 비용이 필요하며, 특히 제한된 목표 기간 안에 제품을 만들기 위해서는 많은 비용이 소요된다.

시스너스를 시작했을 때 우리는 벤처 캐피탈이 결코 우리가 하는 일을 이해하지 못하리라고 생각했고, 그들이 우리를 이해하게 될 시점은 별로 해줄 것이 남아 있지 않은 5년 이후가 될 것이라고 가정했었다. 그러나 다행히도 이러한 우리의 가정은 모두 틀린 것이었다.

1992년 초에 우리의 첫 번째 외부 운영진인 필리페 코트오트Philippe Courtot는 내게 주요 벤처 캐피탈을 소개해주는 데 전혀 시간을 낭비하지 않았다. 나는 그들 모두에게 시그너스의 사업 모델과 기술, 그리고 미래의 목표에 대한 의견을 허울 없이 이야기했고 시그너스를 독립 자산을 가진 회사로 만들기 위해 외부 자본금을 유입하지 않겠다는 최초의 생각도 바꿀 수 있다는 유연한 입장을 취했다. 실제로 시그너스가 독립 자산만으로도 매년 80% 이상씩 성장하는 동안에는 무리 없이 연간 1% 이상의 수익률을 늘려 나갈 수 있을 것이며, 이는 내가 아는 한도에서 매우 좋은 성장 지표인 동시에 우리의 사업이 성숙해간다는 징조였다. 벤처 캐피탈 업계의 유력한 소프트웨어 산업 분석가인 로저 맥나미Roger McNamee의 말은 우리의 성장 가능성을 좀 더 명확하게 나타내 주었다.

> "나는 시그너스의 사업 모델로부터 놀라움과 경탄을 금할 수 없다. 나는 오픈 소스 사업 모델이 효과적으로 적용된다는 사실에 매우 놀랐으며, 그들의 성공을 생각할 때마다 왜 내가 오픈 소스 사업 모델을 그들보다 먼저 생각해내지 못했을까 하는 안타까움을 숨길 수 없다."

외부로부터 자금을 유입하지 않고도 당면한 문제에 자체적으로 대처할 수 있다는

생각이 1996년까지는 전혀 무리가 아니었지만, GNU Pro의 성공을 통해 만들어진 새로운 기회를 현실화하기 위해서는 새로운 계획과 파트너가 필요했다.

우리는 그레이록 매니지먼트Greylock Management와 어거스트 캐피탈August Capital이라는 두 군데의 투자자를 찾을 수 있었다. 그들은 우리가 이루어 놓은 성과와 방법을 잘 이해했으며, 앞으로의 사업 방향과 목표에 대해서도 잘 알고 있었다. 그들은 우리의 계획이 실행될 수 있도록 충분한 자본을 제공해주었는데, 그들이 제공한 625만 달러는 소프트웨어 회사에 대해 1997년 상반기에 이루어진 가장 큰 규모의 투자 였으며 그 집행은 신뢰 속에서 이루어졌다.

샘, 나는 그것들이 싫어. 나는 녹색 계란 프라이와 햄이 싫다구!

— Dr. 수스[44]

기술팀이 열심히 개발에 전념하는 동안에 경영팀은 어떻게 하면 예산을 유용하게 집행할 수 있을까라는 점을 궁리했다. 처음에는 eCos[45]의 설계와 그것을 상용화시키는 데 사용할 사업 모델에 대한 연결 고리를 찾지 못하고 있었다. 기술적인 측면에서 볼 때 하나의 도구로 모든 개발 환경을 설정할 수 있는 유연성과 통일성이 가장 큰 핵심이라는 것을 우리는 잘 알고 있었으며, 사업 측면에서는 이러한 것이 임베디드 시스템 개발에 대한 이윤을 창출할 수 있는 통일된 표준을 만들어 낼 수 있는 방법이라는 것 또한 잘 알고 있었다. 그러나 문제는 누가 우리의 제품을 구입할 것인가라는 점이었다. 기술과 사업이라는 2개의 측면은 1년 반 동안 나름대로의 문제를 가진채 독립적으로 지속되었으며, 그 동안 R&D 비용은 계속해서 묻히고 있었다. 또한 많은 경영진은 오픈 소스의 패러독스를 풀지 못한 채, 두 가지 측면을 결합시키지 못하고 있었다.

마침내 기술팀의 노력이 성공해서 첫 번째 시제품을 선보였을 때, 경영팀은 우리가 임베디드 시스템 분야에 있어 세계 최초의 오픈 소스 아키텍처를 만들었다는 사실을 명확하게 실감할 수 있었다. 개인적으로 그 순간은 내가 GCC를 처음 봤을 때

느낀 흥분을 다시 한 번 느낄 수 있었던 때기도 했다.

오픈 소스는 해커에게 있어 매우 좋은 것이다. 그리고 오픈 소스가 표준을 만들 수 있는 방법은 일반 사용자에게도 좋은 것이다. 그러나 오픈 소스를 이용해서 해커들이 할 수 있는 일과 일반인이 할 수 있는 일에는 명백한 차이가 있다. 우리는 eCos가 해커뿐만 아니라 메인스트림 시장의 일반 임베디드 시스템 개발자도 쉽게 사용할 수 있게 되기를 희망한다. 우리의 목적은 일반 사용자에게 eCos를 통한 고급 수준의 설정 방법을 제공하여 좀 더 쉽게 원하는 목적에 맞는 환경 설정과 사용자화, 그리고 검증 기능을 사용해서 수동으로 수행할 수밖에 없었던 기존의 RTOS 개발자의 수작업을 자동화할 수 있게 하는 것이다. 고급 수준의 도구가 eCos를 소스 코드 차원까지 제어할 수 있게 하고 이러한 도구에 의해 소스 코드가 제어될 수 있도록 설계하여 우리는 최종 사용자가 C나 C++를 이용한 코드를 직접 작성하거나 참조하지 않아도 소스 코드 차원의 작업을 가상으로 수행할 수 있도록 만든 것이다. eCos가 성공할 수 있다는 충분한 확신은 바로 700바이트 정도의 분량을 갖는 최소한의 가상 계층에서부터 인터넷 스택과 파일 시스템을 갖춘 완전한 RTOS의 기능을 갖는 50KB까지 확장될 수 있다는 점이다.

오픈 소스가 단지 가능성일 뿐만 아니라 eCos를 실현할 수 있는 기술적인 실체임을 증명했을 때, 우리는 경쟁 업체보다 10배 이상의 성능을 확보할 수 있었으며, 제품은 출시되자마자 즉각적이고 긍정적인 반응을 얻었다(eCos는 오브젝트 수준의 설정 부분에서 10배 이상의 공간 절약을 제공하며 소스 코드 차원의 접근 방법을 통해서 프로그래머에게 10~100배 정도의 효율을 제공할 수 있다).

GNU에 기반한 우리의 사업 모델을 불가능한 것으로만 생각했던 경우를 돌이켜 본다면, eCos가 시그너스와 임베디드 업계를 위해서 창조할 수 있는 가능성을 충분히 가늠해 볼 수 있을 것이다.

미래에 대한 생각과 비전

이제 오픈 소스 소프트웨어가 기술 관련 자유 시장의 고유한 효율성에 문을 두드리지만, 이는 상당히 조직적이고 예측 불가능한 방법 안에서 이루어지는 것이다. 오픈 소스 사업은 전체 시장을 돕고 미시 경제학을 기반에 둔 조절 기능을 수행하는 애덤 스미스의 '보이지 않는 손'의 역할을 할 수 있으며, 오픈 소스 사업의 성공 여부는 협력을 낳게 한 네트워크 공동체로부터 기술을 성공적으로 이끌어 내면서 사용자의 기술적, 사업적 문제를 함께 풀어 갈 수 있느냐 하는 데 있다.

오픈 소스 소프트웨어에서 만들어진 인터넷은 새로운 오픈 소스 소프트웨어를 만들 수 있는 가장 환상적인 수단이다. 인터넷과 오픈 소스에 좀 더 가까이 다가설수록 우리는 오랫동안 축적된 학문적 지식을 변화시켰던 르네상스와 같은 소프트웨어의 개발과 사용에 대한 변화를 지켜보는 증인이 될 수 있을 것이다. 오픈 소스 소프트웨어가 가진 자유와 함께라면 나는 그 이상 아무 것도 바라지 않는다!

그리하여 그는 새로운 기술에 마음을 쏟았고 그것으로 자연 법칙을 바꾸어 나갔다.

— 제임스 조이스[46]

---

01 역자주_시그너스 솔루션(Cygnus Solutions)은 창립 10주년이 되는 1999년 11월에 세계 최대의 리눅스 배포판 업체의 하나인 레드햇 소프트웨어사에 6억 7천4백만 달러의 금액으로 인수·합병되었으며, 이 글의 저자인 마이클 티만은 레드햇사의 CTO(Chief Technical Officer, 최고 기술 책임자)로 근무하고 있다.

02 역자주_http://www.forbes.com/forbes/98/0810/6203094a.htm 참고.

03 역자주_http://www.amcity.com/sanjose 참고.

04 역자주_http://www.softwaremag.com/98sw500/sm98rnk8.htm#Cygnus 참고.

05 역자주_캘리포니아 기업부의 원래 명칭인 California Department of Corporations는 캘리포니아 지역에 대한 기업 등록과 의료 보험 서비스, 증권, 보험, 금융 서비스, 소비자 보호 등의 업무 허가와 감독 및 입법

을 담당하는 부서다. 이 용어에는 공식적으로 사용되고 있는 번역이 존재하지 않으며 주한 미국 대사관과 캘리포니아 총영사관을 통해서도 적절한 용어를 찾을 수 없었기 때문에 위와 같이 캘리포니아 기업부라고 직역했다.

06 역자주_http://www.gnu.org/manual/emacs-20.3/emacs.html 참고.

07 역자주_doctor 모드는 조셉 와이젠바움이 개발한 ELIZA(엘리자) 프로그램의 내용을 Emacs에서 유사하게 구현한 것으로 ELIZA는 마르타 로저(Martha Roger) 여사가 제안한 모델을 이용해서 만들어진 대화형 구조의 심리 치료 프로그램이다. 와이젠바움은 이 프로그램을 일종의 게임으로 개발했는데, 일각에서 이 프로그램이 출력하는 말들을 심각하게 받아들이는 사람들이 생겨나자 이를 정말로 받아들이지 말라고 해명하고 다녔다는 재미있는 일화가 있기도 하다. doctor 모드로 들어가기 위해서는 Emacs를 실행한 뒤에 메타 키인 Esc 키를 누르고 이어서 x와 doctor를 입력해서 명령어의 전체적인 입력 형태가 'M-x doctor'가 되게 한 뒤에 Enter를 누르면 된다. 원하는 문구를 대화식으로 입력한 뒤에 매번 Enter를 두 번씩 누르면 여기에 맞는 조언을 얻을 수 있다.

08 역자주_dissociated(분리, 뒤죽박죽) 출판 모드를 실행하면 문서의 내용이 단어와 문자를 기준으로 섞이게 되므로 원래의 형태를 상실하게 되지만 재미있는 읽을거리가 생겨나는 이채로운 여흥이 된다. dissociated-press 모드를 실행하기 위해서는 메타 키인 Esc 키를 누르고 x를 누른 뒤에 dissociated-press를 입력해서 전체적인 출력 모양이 'M -x dissocoiated-press'가 되게 한다. Emacs 에디터는 명령행 완성 기능을 제공하기 때문에 dissociated-press를 모두 입력할 필요 없이 dissoci 정도의 단어만 입력한 뒤에 스페이스 키를 연이어 누르면 나머지 문자들이 자동으로 입력된다.

09 역자주_하노이 모드는 3개의 기둥에 쌓아올린 크기가 각각 다른 원형 판들을 큰 판이 작은 판 위로 오지 않도록 하는 원칙을 지키면서 하나의 기둥으로 모두 옮기는 하노이 탑 시뮬레이션 게임이다. 하노이 탑은 컴퓨터 분야에서 재귀 호출을 사용하는 프로그램을 작성할 때 교과서와 같이 등장하는 유명한 내용으로 Esc 키를 누른 뒤에 x와 hanoi를 입력해서 하노이 모드로 들어갈 수 있다. 명령어 입력행에 표시된 형태는'M -x hanoi'다.

10 역자주_MIPS(밉스)는 Million Instructions Per Second의 약자로 1초 동안 실행할 수 있는 명령어의 수를 100만 개 단위로 나타낸 것이다.

11 역자주_아이작 뉴턴(Isaac Newton, 1642~1727), 운동 법칙과 중력을 이용해서 뉴턴 역학이라고 불리는 근대 물리학의 기초를 세웠다. 위 수식은 그의 유명한 저서인 『자연 철학의 수학적 원리(Philosophiae Naturalis Principia Mathematica)』에 기술된 것이다.

12 역자주_빅토르 위고(Victor Hugo, 1802~1885), 19세기 프랑스의 낭만주의 시인으로 『노트르담의 꼽추

(Note-dame de Paris)』, 『레 미제라블(Les Miserables)』과 같은 명작을 남겼다. 위 문구는 1852년에 발표된 『범죄의 역사(Histoire d'un Crime)』의 결론 부분에서 인용된 것이다.

13 역자주_애덤 스미스(Adam Smith, 1723~1790), 중농학파의 입장에서 자본주의 경제 체제를 옹호하며 자유방임에 의한 자율적인 시장 경제를 강조했다. 위 문구는 그의 저서인 『국부론(An Inquiry into the Nature and Causes of the Wealth of Nations)』에서 인용된 것이다.

14 역자주_토머스 제퍼슨(Thomas Jefferson, 1743~1826), 역사학자 랑케가 세계사에서 가장 의미 있는 사건의 하나로 평가한 미국 혁명 당시에 『독립 선언서(Declaration of Independence)』의 기초 내용을 작성했으며 미국 3대 대통령으로 재임하기도 했다.

15 역자주_윌리엄 톰슨 캘빈 경(William Thomson Lord Kelvin, 1824~1907), 영국의 물리학자로 열역학 분야에 지대한 공헌을 했다. SI 단위계의 국제 온도 표준을 나타내는 K(켈빈 온도)는 그의 이름에서 유래한 것이며, 흔히 273.16K를 절대 온도라는 말로 표현한다. 열역학 제2법칙에 대한 논문인 『열의 동역학적 이론에 관하여(On the Dynamical Theory of Heat)』 등의 대표적인 저서가 있다.

16 역자주_토머스 에디슨(Thomas Edison, 1847~1931), 전등과 축음기 등을 발명한 20세기 최고의 발명가로 어린 시절에 달걀을 가슴에 품은 일화와 청각 장애를 딛고 일어선 불굴의 정신으로 유명하다. 그는 1,093개의 기술 특허를 보유하고 있었으며, 그중에 무기(weapon)와 관련된 발명이 1건도 없었다는 점을 스스로 자랑스럽게 생각했다고 한다.

17 역자주_현대의 법은 법의 표현 형식에 따라서 성문법과 불문법의 두 가지 종류로 크게 구분할 수 있다. 성문법은 입법권자가 법의 내용을 명시적인 문서의 형태로 미리 작성한 뒤에 이를 기준으로 법을 적용하는 것을 말하고 불문법은 일정한 제정 절차에 따라서 성문화되지 않은 법을 기준으로 법을 적용하는 것을 말한다. 성문법의 대표적인 예로는 로마의 시민법(civil law)을 근간으로 하는 독일, 프랑스 중심의 대륙법을 들 수 있는데, 우리나라의 경우에는 이러한 대륙법의 영향을 많이 받은 경우에 해당한다. 불문법의 대표적인 예는 판례를 중심으로 한 미국, 영국의 보통법(common law)이며 사건에 대한 판례가 지속해서 축적되어 법 체계를 이룬 형태를 보인다. 보통법은 불문법에 해당하지만 법전 자체가 없는 것을 의미하는 것이 아니라 제정 절차에 의해서 하향식으로 만들어지는 것이 아닌 수많은 세월을 거쳐서 이루어진 판례들이 하나의 상향식 법 체계를 이루는 것을 의미하며, 미국의 인기 TV 연재물이었던 『하버드 대학의 공부 벌레들(The Paper Chase, 1973)』에서 볼 수 있는 법과전문대학원(law school)의 교육 과정처럼 철저한 판례 중심의 적용이 가장 큰 특징을 이루고 있다. 보통법은 모든 소송의 결과가 기존의 법으로 흡수될 수 있으므로 '생활 속의 법'이라는 실리주의적 측면이 강하게 내포되어 있다고 볼 수 있다.

18 역자주_로에 대(對) 웨이드(Roe vs. Wade) 사건은 미혼모의 낙태권을 둘러싼 소송으로, 청원자인 제인

로에(Jane Roe)가 국가를 상대로 벌였던 미국 역사상 가장 미묘한 사건의 하나다(미국의 소송은 청구인과 피청구인의 이름을 사용해서 분류하며, 웨이드라는 이름은 담당 검사였던 헤니 웨이드(Heny Wade)를 가리킨다). 로에 대 웨이드 사건 이외에 피임 실패로 인한 낙태권 인정 여부를 둘러싼 소송이 유사한 시기에 함께 진행되었는데, 대법원은 사생활에 대한 개인의 권리를 보장하는 미합중국 헌법 조항을 근거로 로에의 승소를 판결했다. 1973년에 있었던 로에 대 웨이드 사건은 올해로 26주년이 지났지만 http://www.roevwade.org를 통해서 여성의 사생활 보호를 위한 자유로운 낙태권의 인정과 태아의 생명 존중이라는 치열한 공방이 여전히 계속되고 있다.

19 역자주_ 데이비드 헤켈 월라스(David Henkel-Wallace), 존 길모어(John Gilmore), 마이클 티만(Michael Tiemann)

20 역자주_에스더 다이슨(Esther Dyson, 1951~), ICANN 임시 의장이기도 한 그녀는 빌 게이츠와 함께 미국 컴퓨터 분야에서 최고의 영향력을 가진 사람으로 평가받고 있다. 위 문구는 그녀의 저작인 『Release 2.0: A Design for Living in the Digital Age』에서 인용된 것으로 우리나라에서는 1997년에 경향신문사에 의해서 『인터넷, 디지털 문명이 열린다』라는 제목으로 번역본이 출간된 바 있다. 1998년에 이 책의 개정판인 『Release 2.1』이 출간되었다.

21 역자주_법률 전문 출판사로 중소 규모의 기업을 창업하는 데 필요한 법률 사항들을 자세하게 소개한 책으로 호평을 받았다. http://www.nolo.com 참고.

22 역자주_존 길모어(John Gilmore), 마이클 티만과 함께 시그너스 솔루션사를 공동 창업했으며 GNU 프로젝트에도 많은 기여를 했다. 또한 블루 리본 캠페인으로 유명한 『전자개척재단(http://www.eff.org)』을 만드는 데 공헌했으며, 유즈넷의 alt. 그룹을 만든 장본인이기도 하다. 그는 1999년에 자유 소프트웨어 재단이 수여한 제2회 자유 소프트웨어 재단상의 후보로 지명되기도 했다.

23 역자주_손자(孫子, Sun Tzu, BC. 500), 위 문구는 중국 춘추 전국시대 때의 사상가인 손자의 『병법(孫子兵法, The Art of War)』 군형편(軍形篇)에서 인용된 것으로 전문(全文) 내용과 뜻은 다음과 같다. 「孫子曰 昔之善戰者 先爲不可勝 以待適之可勝(손자왈 석지선전자 선위불가승 이대적지가승, "손자가 말하기를 예전에 용병을 잘하는 자는 먼저 적군이 아군을 이길 수 없도록 완벽한 태세를 갖춘 뒤에 아군이 적군을 이길 수 있을 때를 기다렸다.")」 손자의 병법에서 가장 널리 알려진 지략인 "나를 알고 적을 알면 백번 싸워도 결코 위태롭지 않다."는 뜻의 「知彼知己 百戰不殆(지피지기 백전불태)」는 모공편(謀攻篇)에 실려 있다.

24 역자주_EDS는 세계 최대의 시스템 통합(System Integration) 및 컨설팅 전문업체며 우리나라에는 1987년도에 현재의 LG와 합자 설립한 LG-EDS가 다양한 분야에서 활동하고 있다. 대법원의 판례들을 데이터베이스로 구축한 뒤에 이를 웹으로 서비스하는 경우를 생각해 보면, 생소한 관련 부서를 대법원 안에 직접 만

드는 것보다는 해당 업무를 외부 전문 업체에 위탁해서 운영하는 것이 비용과 효율성을 좀 더 높이는 방법이 되는데 이러한 운영 방식을 아웃소싱(out sourcing)이라고 한다. 아웃소싱은 시스템 운영, 네트워크 관리, 애플리케이션 개발 등은 물론이고 자사의 하드웨어와 내부 인력을 아웃소싱 업체에 함께 이관하는 방법을 통해서 인력과 비용을 합리화시킬 수 있다는 장점을 갖고 있다.

25 역자주_역자주 43 참고.

26 역자주_역자주 43 참고.

27 역자주_리처드 스톨만이 쓴 『GNU 운영체제와 자유 소프트웨어 운동』의 '첫 번째 단계' 부분에서 언급된 것과 같이 GCC 컴파일러는 CISC 아키텍처인 모토로라 68000 시스템(m68k)에 맞게 개발된 것이었기 때문에 RISC 아키텍처로 최적화해서 이식시키기 위해서는 새로운 기능들이 추가되어야 했다. 컴퓨터는 CPU의 마이크로 명령어 구성 방식에 따라서 CISC(시스크, Complex Instruction Set Computer)와 RISC(리스크, Reduced Instruction Set Computer)의 두 가지 형태로 크게 분류할 수 있다. C나 포트란과 같은 프로그래밍 언어로 작성된 프로그램은 컴파일 과정을 거쳐서 CPU에서 직접 실행될 수 있는 기계어로 번역되는데, 만약 고급 언어의 특정 기능을 수행할 수 있는 명령어를 CPU 차원에서 직접 지원하게 되면 컴파일 과정이 단순해지고 시스템의 전체적인 수행 능력을 향상할 수 있게 된다. 이러한 장점을 높이기 위해서 고급 언어의 기능을 수행할 수 있는 명령어를 가능한 한 많이 포함해서 설계한 시스템이 CISC다. 그러나 명령어의 수가 많아지면 CPU의 내부 회로와 명령어 구성 방식이 복잡해져서 실행 속도가 늦어지는 단점이 발생하게 된다. RISC 아키텍처가 고안된 가장 큰 이유는 명령어의 수를 가능한 최소로 줄이고 모든 연산이 CPU 안에서만 수행될 수 있도록 해서 실행 속도를 높이기 위한 것이다. CISC와 RISC 아키텍처는 컴퓨터의 수행 능력과 컴파일 과정을 최적화시키려는 의도와 실행 속도를 최대화하려는 두 가지 목적에 따라서 만들어진 것으로 시스템 설계의 측면에서 볼 때 하나의 시스템 안에 모든 기능을 최적화시켜서 구현할 수는 없기 때문에 한 가지를 충족시키면 다른 한 가지는 희생할 수밖에 없는 선택을 해야 한다. 이처럼 특정한 부분의 최적화를 위해서 다른 부분의 성능을 어느 정도 희생시키는 것을 트레이드 오프(trade-off)라는 용어로 표현한다. 대표적인 CISC 아키텍처로는 DEC의 VAX와 IBM 370 그리고 인텔의 80x86이 있다. 시그너스 솔루션이 GCC 상용 패키지를 본격적으로 개발하기 시작한 1989년은 인텔이 1985년과 1988년에 각각 발표한 386 DX와 386 SX에 이어서 486 프로세서를 발표함으로써 32비트 CISC 아키텍처에 의한 개인용 컴퓨터 시대의 전성기가 시작되던 시기에 해당한다. 또한 이 시점은 리눅스의 모태가 된 PC용 유닉스 운영체제인 미닉스가 인터넷을 통해서 전 세계적으로 폭넓게 확산되던 때였으며, 17가지 종류 이상이 난립하던 상업용 PC 컴파일러 시장을 분당 7, 000라인이라는 당시로써는 경이로운 컴파일 속도로 처리했던 볼랜드사의 Turbo C/C++가 석권하던 때이기도 하다. 그러나 호환성이라는 고육지책에 의해서 다른 아키텍처로 돌아설 수 없을 만큼 팽창되어버린 개인용 PC 이외의 시장에서는 실행 속도를 극대화하기 위해서 CISC

에서 RISC로 아키텍처의 주력이 변경되었는데, 1984년 이후부터 새롭게 등장한 MIPS와 Sparc, HPPA-RISC, PowerPC, Alpha 등의 프로세서는 모두 RISC 아키텍처에 해당하며 1995년부터 본격적으로 판매된 펜티엄 프로세서에는 RISC 기술이 부분적으로 도입되었다.

28 역자주_GNU C++ 컴파일러는 C 언어 컴파일러인 GCC에 C++ 확장 패키지를 추가해서 구성하는 방법을 사용한다. C++ 컴파일러는 C++ 또는 G++이라고 부르며 컴파일에 사용되는 명령어 또한 C++과 G++ 두 가지 형태를 모두 사용할 수 있다. http://www.gnu.org/software/gcc/gcc.html 참고.

29 역자주_요기 베라(Yogi Berra, 1925~), 미국 메이저리그의 전설적인 야구 선수로 여러 분야에서 인용되고 있는 많은 말을 남겼다. "Never give up, because it ain't over until it's over.", "Remember that whatever you do in life, 90% of it is half mental.", "You can't think and hit at the same time.", "If you can't imitate him, don't copy him." 등의 의미깊은 말들을 모아놓은 그의 어록이 『Yogi Book』이라는 이름으로 출판되어 있지만, 우리나라에는 소개되어 있지 않다.

30 역자주_CVS는 Concurrent Versions System의 약자로 프로그램이나 문서화 작업의 버전 및 갱신 과정을 자동으로 관리하는 프로그램이다. 예를 들어 장기간의 시일이 소요되는 프로그래밍 작업을 여러 사람이 공동으로 수행할 경우에 프로그램의 소스가 수정되는 과정에서 버그가 발생하면 이를 발견하기가 쉽지 않다는 단점을 갖게 된다. 그러나 CVS를 사용하면 수정이 이루어진 각각의 버전을 좀 더 쉽게 분석할 수 있기 때문에 효율적인 진행이 가능해진다. 현재 대부분의 오픈 소스 프로젝트들은 CVS를 이용해서 진행하는 것이 일반적이며 프로그램뿐만 아니라 문서 번역 프로젝트 등의 다양한 작업에 유용하게 사용될 수 있다. CVS와 CVS 사용에 대한 상세한 설명이 담긴 매뉴얼은 http://www.gnu.org/software/cvs/cvs.html을 통해 참고할 수 있다.

31 역자주_커버로스(Kerberos)는 네트워크를 통한 인증 시스템의 한 종류이다. telnet을 통해서 다른 시스템으로 로그인하는 경우를 생각해보면 입력한 사용자 이름과 패스워드가 네트워크를 통해 ASCII 상태 그대로 전달되는 형식을 가지기 때문에 이 과정에서 자신의 사용자 이름과 패스워드가 유출될 위험성이 존재한다. 따라서 telnet 과정에 사용되는 패스워드 자체를 처음부터 암호화해서 전달하게 되면 유출 위험성을 없앨 수 있게 되는데, 이러한 용도로 사용되는 것이 커버로스 등의 네트워크 인증 시스템이며 동일한 용도로 사용되는 프로그램으로 SSH(Secure SHell)가 있다. 커버로스와 SSH, PGP 등의 암호화 도구들은 암호화 기술 자체를 전략 무기로 간주하는 미국 연방법에 따라 외국으로의 수출이 철저하게 규제되고 있었지만, 전자상거래의 보안 신뢰도를 높이기 위해 윈도우 2000에 암호화 기술을 내장하려고 했던 마이크로소프트와 관련 업계의 로비 때문에 관련법이 완화되기에 이르렀다. 그동안 PGP 등의 암호화 프로그램이 한국에서도 사용될 수 있었던 이유는 독점 특허나 특정 국가의 법률에 따라 규제되던 기술이라 하더라도 일단 해당 기술의 내용이 일반적으로 널리 알려지게 되면 별다른 제한 없이 사용할 수 있다는 국제 저작권법상의 해석에 따라

미국 밖으로 불법 유출된 프로그램의 소스 코드를 이용해서 국제 버전으로 제작된 프로그램을 사용할 수 있었기 때문이다. 커버로스에 대한 좀 더 자세한 정보는 http://www.pdc.kth.se/kth-krb를 통해 참고할 수 있다.

32 역자주(표 전체)_표에 포함된 바이너리 도구들은 컴파일러와 디버거를 제외하고 모두 binutils 패키지 안에 포함되어 있으며, 실행 파일의 이름은 차례대로 as, ar, ld, size, strip, nm이다. GNU C 컴파일러는 gcc 또는 egcs라는 이름으로 배포되며 디버거의 패키지 이름은 gdb다.

33 역자주_컴파일러나 어셈블러에 의해 생성되는 객체 파일과 실행 파일의 포맷은 최초의 유닉스에서 사용되던 A.OUT(Assembler.OUTput)부터 시작해서 CPU의 아키텍처와 효율성 및 호환성을 기준으로 다양하게 발전했는데, 특정한 CPU 아키텍처 환경에서 생성된 바이너리 파일은 다른 아키텍처에서 실행될 수 없는 고유한 특성을 갖게 된다. 현재 리눅스에서는 유닉스 시스템 V Release 4부터 도입되기 시작한 ELF(Excutable and Linkable Format)를 기본 바이너리 포맷으로 사용한다. ECOFF와 XCOFF는 유닉스 시스템 V Release 3에서 사용된 COFF(Common Object File Format)의 확장 포맷이며, IEEE695는 IEEE에서 표준화한 IEEE Standard for Microprocessor Universal Format for Object Modules 포맷을 의미한다.

34 역자주_GNUpro 툴킷은 리눅스를 포함한 유닉스용과 윈도우 95/98/NT용이 모두 개발되어 있으며, http://www.cyg nus.com/pubs/gnupro를 통해 자세한 정보를 참고할 수 있다.

35 역자주(표 전체)_복합 연평균 성장률(Compound Annual Growth Rate, CAGR)은 사업 초기부터의 성장률을 연도별로 나타낼 수 있는 지표이며 공식 [{(마지막 연도 총액/시작 연도 총액) ^ (1/연도 수)} - 1] * 100을 통해 산출할 수 있다. 따라서 1991년의 CAGR은 1991년과 1990년의 수치를 사용해서 [{(1500000/725000) ^ 1}-1] * 100 = 106%가 되고, 1993년의 CAGR은 시작 연도인 1990년과의 수치를 이용해서 [{(4800000/725000) ^ (1/3)} - 1] * 100 = 87%가 된다.

36 역자주_알렉산더 그레이엄 벨(Alexander Graham Bell, 1847~1922), 위 문구는 1876년 3월 10일에 성공했던 벨과 그의 조수 왓슨 사이에 이루어진 세계 최초의 전화 통화 내용이다. 위 통화 내용은 인간의 음성을 전파로 이동시킬 수 없다는 고정 관념을 불식시키고 새로운 커뮤니케이션 문명의 시작을 가능하게 한 역사적인 사건이었지만, 벨이 전화를 발명하게 된 최초의 동기는 청각 장애를 가진 그의 아내(Mabel Hubbard)를 위한 애틋한 사랑의 발로에서였다.

37 역자주_여기에 언급된 소프트웨어에 대한 자세한 정보는 http://www.gnu.org/software에 포함된 개별적인 개발 프로젝트를 통해 참고할 수 있다.

38 역자주_시그너스 솔루션의 고객층은 GNUpro 등을 이용해서 프로그램을 만드는 개발업체라는 사실을 다시

한 번 상기하기 바란다. 따라서 시그너스의 제품을 구매하거나 개발 용역 계약을 체결할 경우에는 계약에 의한 결과물과 이익이 오픈 소스로 다시 환원되어 축적된 기술이 경쟁업체에 제공될 수 있기 때문에 구매자의 입장에서는 '구매 결정 가능성'과 같은 문제가 제기될 수 있다.

39 역자주_컨설팅 전문업체 캐즘 그룹(The Chasm Group)의 설립자인 조프리 무어(Geoffrey Moore)의 첫 번째 저서인 『캐즘을 넘어서(Crossing Chasm)』는 첨단 기술 분야에서 존재하는 초기 시장과 메인스트림 시장 사이의 두꺼운 벽을 극복하기 위한 마케팅 전략에 대해서 논파하는 책이다. '틈' 또는 '협곡'을 의미하는 단어인 캐즘은 첨단 기술 관련 신생 기업이 성공하기 위한 기술적 장점을 통해 얻을 수 있는 초기 시장의 수요와 대기업에 의해 선점된 주력 시장인 메인스트림 사이의 벽을 의미하며, 조프리 무어는 이 책에서 그러한 장벽을 극복하는 방법들을 마케팅 차원에서 설명한다. 이 책은 초판이 출간된 1991년 이래로 미 국 내 많은 대학에서 경영, 마케팅 관련 교재로 사용되고 있으며, 우리나라에서는 1997년에 세종서적에 의해서 『벤처 마케팅』이라는 조금은 색다른 제목으로 번역되었다.

40 역자주_격주간으로 발행되는 세계적인 경제지 포천(Fortune magazine)이 선정하는 500대 기업 중에서 상위 100개 기업을 지칭한다. http://www.pathfinder.com/fortune/fortune500/index.html 참고.

41 역자주_Original Equipment Manufacture의 약자로 생산된 제품에 주문자의 상표를 부착해서 납품하는 일종의 하청 또는 협력 형태의 생산 방식을 의미한다.

42 역자주_IT(Information Technology) 업계 시장조사 전문기관. http://www.idc.com

43 역자주_RTOS(Real Time Operating System, 실시간 운영체제)란 운영체제의 종류를 의미하는 용어가 아니라 운영체제의 실시간 처리 기능을 강조한 개념이다. 따라서 요청된 작업에 대한 처리를 실시간으로 수행할 수 있는 리눅스도 RTOS라고 할 수 있다. RTOS를 이해하기 위해서는 다음과 같은 임베디드 시스템의 개념을 함께 이해하는 것이 도움이 된다. 컴퓨터 이외에 전자수첩이나 통신 기기, 자동 세탁기, 셋톱박스와 같은 가전제품 안에는 리눅스나 윈도우 98/NT와 같은 운영체제는 포함되어 있지 않지만 자동으로 작동되는 자체적인 운영 환경을 유지하기 위해서 독립된 운영체제나 애플리케이션 프로그램이 내장되어 있는데 이러한 형태의 제품을 임베디드 시스템(embedded system) 또는 독립 장비라는 말로 표현한다. 일반적으로 임베디드 시스템은 속도와 크기, 처리상의 효율성을 위해서 입출력 장치와 같은 주변 장치를 갖지 않고 전원을 올림과 동시에 ROM에 내장된 애플리케이션이 곧바로 실행되어 사용자의 요구를 실시간으로 처리하는 구조로 되어 있다. 기본적으로 PC를 제외한 가전제품과 통신 기기 등은 모두 임베디드 시스템이라고 할 수 있으며, 에어컨이나 비디오와 같은 일상적인 가전제품의 작동 방법과 구조를 생각해 본다면 임베디드 시스템의 개념을 좀 더 쉽게 이해할 수 있을 것이다. 임베디드 시스템에 내장되거나 임베디드 시스템에 사용할 애플리케이션을 개발할 수 있는 환경을 제공하는 것이 RTOS이며, 임베디드 시스템용 애플리케이션을 개발

하기 위해서는 호스트(host)와 타깃(target)이라는 2개의 작업 환경을 사용하게 된다. 네트워크 연결을 통해서 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 에디터, 컴파일러, 디버거 등의 개발 도구들이 포함될 수 없기 때문에 개발 환경을 제공할 수 있는 좀 더 큰 규모의 시스템이 필요한데, 이러한 용도로 사용되는 개발 모체를 호스트라는 말로 표현하고 호스트에서 만들어진 프로그램이 실제로 실행되는 최종 플랫폼을 타깃이라고 한다. 임베디드 시스템을 개발할 때는 호스트와 타깃이 별개의 아키텍처로 선택될 수 있으므로 애플리케이션의 개발은 인텔 80x86 아키텍처에서 이루어지지만 애플리케이션이 실행될 실제 타깃은 모토로라의 PowerPC나 ARM으로 선택하는 등의 이질적인 아키텍처 조합이 가능해진다. GNUpro 툴킷이 175개 조합의 호스트/타깃 플랫폼을 지원할 수 있다고 언급된 부분에서 사용된 호스트/타깃은 바로 이러한 의미며 호스트와 타깃이 하나의 동일한 시스템을 지칭할 경우에는 네이티브(native)라는 용어를 사용하게 된다. 따라서 인텔 펜티엄 프로세서가 장착된 호스트 컴퓨터에 리눅스를 설치한 뒤에 GCC 컴파일러로 네이티브 코드를 생성했다면 이때 생성된 실행 파일은 컴파일 호스트와 동일한 타깃 플랫폼인 펜티엄 시스템에서만 실행할 수 있는 코드가 된다. 그러나 임베디드 시스템을 만들기 위해서는 특정 호스트에서 컴파일된 바이너리가 다른 아키텍처의 타깃에서 실행될 수 있어야 하는데, 이러한 형태로 바이너리를 생성해줄 수 있는 컴파일러를 크로스 컴파일러(Cross Compiler)라고 한다. 크로스 컴파일러는 --host와 --target 옵션을 설정해서 GCC 소스 코드를 컴파일하는 방법으로 만들 수 있으며 http://www.objsw.com/CrossGCC를 통해서 관련 정보를 모두 참고할 수 있다. 이번 장의 소주제인 eCos는 임베디드 시스템 개발을 지원하기 위해서 시그너스가 개발한 RTOS다. http://www.cygnus.com/ecos 참고.

44 역자주_닥터 수스(Dr.Seuss, Theodor Seuss Geisel, 1904~1991), 위 문구는 1960년에 미국에서 출판된 유아용 읽기 교재인 『Green eggs and Ham』에서 인용된 것으로 원문의 내용은 "I do not like them, Sam-I-am. I do not like green eggs and ham."이다. 이 책의 전체적인 내용은 주인공격인 Sam-I-a m이 가상의 음식인 녹색 계란 프라이와 햄을 또 다른 등장인물에게 먹어볼 것을 권유하지만, 녹색 계란과 햄은 싫다고 거부하는 단순한 대화를 60페이지 분량으로 구성한 것이다. 그러나 이 책은 단순한 내용과 최소한의 단어들을 마치 최근의 유아용 TV 프로그램인 꼬꼬마 텔레토비처럼 반복과 율격을 살린 형태로 구성해서 3~6세 사이의 유아들이 자연스럽게 읽기와 언어 인식 효과를 높일 수 있게 되어있다. Sam-I-am이라는 이름도 "내 이름은 샘이에요."라는 문장인 "I am Sam."을 책의 처음에 제시했다가 이를 다음 단락에서 도치시켜서 Sam-I-am으로 만든 것이며, 원문인 "I do not like them, Sam-I-am. I do not like green eggs and ham."도 자연스러운 율격과 각운을 갖추게 되어 유아들이 읽기에 흥미를 느낄 수 있게 되어 있다. 이 책이 사용하는 대표적인 반복과 확장의 방법은 다음과 같다. "I do not like them in a box. I do not like them with a fox. I will not eat them in a house. I do not like them with a

mouse. I do not like them here or there. I do not like them any where! I do not like green eggs and ham! I do not like them, Sam-I-am." 책의 마지막 부분에서는 겉모습만 보고 계속해서 싫다고 우기던 녹색 계란과 햄을 결국 먹어보게 된 후 에이를 좋아하게 되어, 이전의 부정문들이 모두 긍정문으로 다시 한 번씩 반복되는 구조로 되어있다. 따라서 여기서 인용된 "나는 그것들이 싫어. 나는 녹색 계란 프라이와 햄이 싫다구."라는 문장은 "시도해보기도 전에 겉모습만 보고 판단하는 것은 좋지 않다."라는 의미를 담았다고 할 수 있다. 닥터 수스의 책은 50여 년이 지난 오늘날까지도 그의 책만으로 시간표를 구성한 '닥터 수스의 날' 행사를 초등학교에 포함할 만큼 전 국민적인 사랑을 받는 미국 내의 대표적인 유아용 교재다.

45 역자주_eCos는 Embedded Cygnus Operating System의 약자다.

46 역자주_제임스 조이스(James Joyce, 1882~1941), 위 문구는 1916년에 출판된 제임스 조이스의 소설『젊은 예술가의 초상(A Portrait of the Artist as a Young Man)』의 도입 부분에서 인용된 것으로 실제로는 제임스 조이스가 아닌 로마의 시인 오비디우스(Publius Ovidius Naso)의 대서사시인『변신(Metamorphoses)』제8장에서 재인용된 것이다. 우리나라에서 번역된 조이스의 책에서는 arts를 예외 없이 예술이라고 번역하고 있지만『변신』에서 이 문장이 기술되는 부분은 이카로스의 아버지인 다이달로스가 아들과 함께 미궁을 빠져나가기 위해서 밀랍으로 날개를 만드는 장면에 해당하기 때문에 티만의 글에서는 arts를 '기술' 또는 '새로운 발명품'이라고 번역하는 것이 더욱 타당할 듯싶다. 제임스 조이스는 의식의 흐름이라는 기법을 사용해서 20세기 영문학에 새로운 지평을 연 작가로『율리시스(Ulysses)』,『더블린 사람들(Dubliners)』,『피네간의 경야(Finnegans Wake)』등의 작품을 썼으며, 특히 호머의 오디세이에서 모티브를 가져온 율리시스는 20세기 영문학의 최대 걸작 중 하나로 평가받는다. 의식의 흐름 기법은 우리나라의 대표적인 초현실주의 시인인 이상의『오감도』와『날개』등에 사용되기도 했다.

6 | 소프트웨어 공학

폴 빅시Paul Vixie

송창훈 역

소프트웨어 공학[01]은 프로그램을 만드는 단순한 작업 이상의 다양한 측면들을 갖고 있는 광범위한 분야이다. 그러나 많은 오픈 소스 프로젝트는 소프트웨어 공학적인 측면을 고려하지 않은 채 프로그램을 만들어 배포하는 방식을 취하며, 그렇게 해도 별다른 문제 없이 대중적인 소프트웨어를 만들 수 있다는 것은 지금까지의 실례를 통해서 명백하게 알 수 있는 점이다. 나는 이 글을 통해서 소프트웨어 공학의 일반적인 요인들과 이와 대응되는 오픈 소스의 측면을 비교함으로써 소프트웨어 개발에 대한 접근 방식의 차이점과 그러한 차이가 갖는 의미들을 설명하고자 한다.

소프트웨어 공학 공정

소프트웨어 공학의 공정은 일반적으로 다음과 같은 요소[02]로 이루어진다.

- 요구 사항 분석
- 시스템 디자인
- 세부 디자인
- 구현
- 통합
- 필드 테스트
- 사후 지원

이와 같은 공정 단계는 순차적인 구성을 갖기 때문에 하나의 단계가 완성되어야만 비로소 다음 공정으로 넘어갈 수 있으며, 하나의 공정이 수정된 경우에는 이와 관련된 모든 공정이 재검토되거나 수정되어야 하는 구조를 갖는다. 경우에 따라서는 여러 개의 모듈이 완전히 기술되기 이전에 이와 연관된 다음 모듈을 기술하거나 구

현하는 것도 가능한데, 이러한 형태를 사전 개발advanced development 또는 사전 조사advanced research라고 부른다.

소프트웨어 공학 공정의 모든 요소에는 등위 검토peer review와 멘토mentor 및 관리 검토, 제휴 검토cross-disciplinary와 같은 검토 방법이 필수적으로 포함되어야 한다. 등위 검토는 비슷한 수준과 역할을 가진 사람이 프로그램의 소스 코드를 분석하는 등의 방법을 통해서 세부적인 사항들을 검토하는 것이며, 제휴 검토는 몇 가지 검토 방법을 함께 사용하는 것이다. 또한 멘토mentor 및 관리 검토는 일종의 컨설팅과 같이 상위 접근 능력을 가진 측면에서의 검토 방법을 의미한다.

또한 소프트웨어 공학의 구성 요소에는 문서 자료와 소스 코드를 불문하고 모두 버전 표시와 감사(監査) 내역이 포함되어 있어야 한다. 어떠한 요소에 대한 수정 여부를 결정하기 위해서는 몇 가지 형태의 검토가 필요하며 검토의 정도는 수정 범위와 직접 연관해서 고려되어야 한다.

요구 사항 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 가는 것이다.

'요구 사항 분석서Marketing Requirements Document, MRD'는 "무엇을 개발할 것이며 개발된 소프트웨어를 누가 사용할 것인가?"라는 사항을 결정하는 문서다. 실패한 많은 프로젝트는 구태의연한 기존의 관행을 답습해서 마케팅 차원으로 만들어진 요구 사항 분석서가 엔지니어링 단계로 단순히 전달되므로, 실무를 담당할 사람들이 정확한 내역을 파악하지 못해 그러한 제품을 만들지 못하리라고 불평하게 된다. 요구 사항 분석서는 마케팅과 엔지니어링의 공동 노력에 의해서 만들어지는 것이며 이런 작업에는 많은 양의 문서를 작성하고 검토하는 일이 함께 수반되어야 한다.

시스템 디자인

시스템 디자인이란 제품을 '모듈(때로는 '프로그램')'의 형태와 이러한 모듈 사이의 상호 관계에 대한 형태로 기술하는 상위 개념의 디자인을 의미한다. 시스템 디자인 단계에서 작성되는 문서인 '시스템 디자인 문서System Design Document, SDD'의 목적은 우선 정상적으로 작동되는 제품을 만들 수 있다는 신뢰성을 확보하기 위한 것이고, 둘째로 소프트웨어를 만들면서 수행될 작업의 총량을 측정할 수 있는 기반을 만드는 것이다. 시스템 디자인 문서에는 고객의 요구와 제안된 시스템 디자인에 의해서 그러한 요구가 충족되었는지를 검토할 수 있는 검증 계획이 포함되어야 한다.

세부 디자인

세부 디자인은 시스템 디자인 문서에서 기술된 모든 모듈을 구체적으로 기술한 문서인 '세부 디자인 문서Detailed Design Document, DDD'를 만드는 단계다. 이 단계에서 소프트웨어의 실행에 따른 명령행 형식과 API 호출 방법, 외부에서 참조할 수 있는 자료 구조 등과 같은 모듈 사이의 인터페이스와 의존성이 모두 결정된다. 세부 디자인으로부터 PERT 차트나 GANT 차트[03]와 같은 두 가지 종류의 차트가 만들어져서 어떤 일이 어떤 순서로 수행되어야 하는지와 각각의 모듈이 완성되는 데 소요되는 시간에 대한 좀 더 정확한 예측이 이루어질 수 있다.

모든 모듈에는 모듈을 구현할 사람에게 해당 기능을 검증해 볼 수 있는 검사 항목과 유형을 알려줄 수 있는 단위 검사 계획unit test plan이 포함되어야 한다. 모듈의 기능을 검사하는 것 이외의 방법으로 수행되는 단위 검사를 사용할 수도 있는데, 여기에 대해서는 뒷부분에 있는 '구현'과 '테스트 세부 사항'에서 설명하기로 한다.

구현

세부 디자인 문서에 기술된 모든 모듈은 빠짐없이 구현되어야만 한다. 구현 단계에서는 소프트웨어 공학 공정의 가장 핵심 부분이라 할 수 있는 프로그램의 코딩 작

업이 실제로 이루어진다. 프로그램 코딩은 소프트웨어 공학에 있어서 혼자서도 효과적으로 배울 수 있는 유일한 부분이기 때문에 때때로 소프트웨어 공학에서 가르치거나 학습하는 유일한 부분이 되기도 한다. 그러나 프로그램 코딩은 소프트웨어 공학을 구성하는 일부분의 하나이므로 이는 매우 유감스러운 일이다.

하나의 모듈이 만들어진 뒤에는 다른 모듈이나 시스템 차원의 검사 공정에 의해서 성공적으로 검사되고 사용된 이후에야 비로소 구현된 것으로 간주될 수 있다. 모듈을 만드는 과정은 소스 코드를 작성하고 컴파일하는 작업을 반복하는 전통적인 방법을 따른다. 모듈을 검사하는 방법에는 세부 디자인 단계에서 기술되었던 모듈 단위의 기능 검사functional test와 회귀 검사regression test가 포함되며, 이외에도 성능 검사performance test와 압박 검사stress test, 그리고 코드 커버리지 분석 등이 포함된다. 압박 검사는 검사의 한계를 극대화하는 방법이며, 회귀 검사는 수정이나 확장이 이루어질 때마다 이미 검증되었던 모듈이나 프로그램을 다시 검사해서 오류 여부를 재확인하는 것이다. 코드 커버리지 분석에 대해서는 '테스트 세부 사항'에서 설명하기로 한다.

통합

모든 모듈이 명목상으로 완성되었다면 시스템 차원의 통합이 이루어질 수 있다. 이 과정에서 모든 모듈을 하나의 소스 코드 풀pool로 이동시킨 뒤에 컴파일과 링크 과정을 거쳐서 하나의 단일 시스템으로 만들게 된다.

통합 작업은 다양한 종류의 단위 모듈을 구현하는 작업과 함께 병행해서 점진적으로 이루어질 수도 있지만, 모든 모듈이 확실하게 완성되기 전까지는 통합 공정의 종결에 대한 섣부른 결정이 이루어져서는 안 된다.

통합 공정에는 시스템 차원의 검사 방법을 개발하는 것도 포함된다. 완성된 패키지가 설치되는 형태를 가지는 경우에는(단순히 tar 파일을 풀거나 CD-ROM에서 파

일을 복사하는 형태를 의미할 수도 있다) 전용 시스템이나 시뮬레이션 환경에서 이러한 기능을 검사해볼 수 있는 자동화된 방법이 필요하다.

때때로 미들웨어 분야에서는 패키지가 단지 소스 코드를 빌드한 형태일 수도 있기 때문에 이러한 경우에는 설치나 설치 도구에 대한 확인 없이 빌드된 소스 풀에서 시스템 검사가 이루어질 것이다.

통합된 패키지가 설치 가능한 형태였을 경우에는 일단 패키지가 설치된 후에 자동화된 검사 공정에 의해서 사용 가능한 모든 명령어와 엔트리 포인트가 조합 가능한 인수와 함께 테스트될 수 있어야한다. 만약 시스템이 데이터베이스 등을 생성하는 것이라면 자동화된 검사 도구는 테스트용 데이터베이스를 생성한 후에 별도로 작성된 외부 접근 도구를 이용해서 데이터베이스의 무결성 여부를 검증해 볼 수 있어야 한다. 이러한 과정에 단위 검사가 활용될 수 있게 하는 것도 가능하며 통합과 빌드, 패키지 공정 중에 모든 단위 검사가 순차적으로 수행되도록 할 수도 있다.

필드 테스트

필드 테스트field test는 일반적으로 내부에서부터 시작된다. 이 말은 소프트웨어 패키지를 만든 기관의 직원이 그들의 컴퓨터에서 소프트웨어를 먼저 테스트해본다는 의미며, 여기에는 데스크톱 컴퓨터와 노트북, 서버와 같은 모든 종류의 시스템 사양에 대한 테스트가 포괄된다.

고객에게 새로운 소프트웨어나 새로운 버전의 소프트웨어를 사용해보기를 권할 때 자신 있게 '우리가 사용하고 있는 시스템'이라는 말을 할 수 있기를 원할 것이다. 따라서 내부 필드 테스트는 이러한 관점에서 볼 때도 중요한 것이다. 내부 필드 테스트 과정 동안에는 소프트웨어 개발자들의 기술 지원이 직접 제공될 수 있어야 한다. 내부 필드 테스트는 보통 알파 테스트alpha test라는 말로 표현되기도 한다.

궁극적으로는 고객이나 예상 고객의 컴퓨터에서 소프트웨어를 실행해보는 외부 필드 테스트가 필요하다. 외부 테스터들은 내부 테스터와 비교할 때 소프트웨어의 사용에 대한 다른 습관과 방법을 갖기 때문에 사소하고 명백한 것을 포함해서 많은 결함을 발견할 수 있다. 따라서 외부 필드 테스트에는 피드백을 가능한 많이 제공해줄 수 있는 친숙한 고객을 참여시키는 것이 바람직하다. 소프트웨어 개발자는 외부 필드 테스트 기간 중에도 단계적인 방법에 의해서 외부 테스터와 연결될 수 있어야 한다. 외부 필드 테스트는 일반적으로 베타 테스트beta test라는 말로 표현되기도 한다.

필드 테스트 과정에서 발견된 결함은 선임 개발자와 기술 마케터에 의해서 우선순위가 정해진 뒤에 문서상으로 수정될 수 있는 것은 무엇이며, 현재의 버전이 릴리즈되기 전에 수정돼야 할 부분과 다음 버전에서 수정돼야 할 부분은 무엇이며, 또한 수정되지 않아도 될 부분은 무엇인가 등의 사항이 결정되어야 한다.

사후 지원

필드 테스트 과정이나 소프트웨어가 배포된 이후에 발견된 결함들은 모두 추적 시스템에 기록되어야 한다. 결함에 대한 사항은 시스템의 정의나 문서에 대한 수정 또는 모듈의 정의나 구현에 대한 수정을 맡을 소프트웨어 엔지니어에게 최종적으로 맡겨져야 한다. 결함이 수정된 뒤에는 최초의 결함과 그것이 해결되었음을 확인할 수 있는 단위 검사 공정과 시스템 차원의 검사 공정이 회귀 검사의 형태로 포함되어야 한다.

요구 사항 분석서MRD가 엔지니어링과 마케팅 사이의 모험적인 결합인 것처럼 사후 지원 또한 엔지니어링과 고객 서비스 사이의 모험적인 결합이라고 할 수 있다. 이러한 모험의 성공 여부에 대한 관건은 버그 리스트를 짜임새 있게 작성하는 것과 특정한 버그들을 체계적으로 분류하는 것, 그리고 판매 직전의 소프트웨어 릴리즈에 포함될 치명적인 결함을 최대한 줄이는 것 등이다.

테스트 세부 사항

코드 커버리지 분석

코드 커버리지 분석code coverage analysis[04]은 선행 처리기나 오브젝트 코드 수정기 또는 컴파일러나 링커의 특별한 모드를 이용해서 프로그램 코드의 운영 방법을 검토하는 것으로 이루어진다. 소스 코드의 블록 안에서 조합할 수 있는 실행 경로를 코드 경로codepath라고 하는데, 코드 커버리지 분석은 조합 가능한 모든 코드 경로를 파악해서 실행 중에 어떠한 코드 경로가 실행되었는지를 확인하고 기록할 수 있어야 한다. 따라서 코드 커버리지 분석을 이용하면 좀 더 효율적인 코드로 최적화하는 것이 가능해진다.

다음과 같은 전형적인 C 코드를 살펴보자.

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

만약 error 변수가 초기화되지 않은 상태에서 2행이 실행된다면 어떤 값인지 모르는 변수 값이 증가되는 결과가 되므로 이후의 프로그램은 정의되지 않은 예측 불가능한 결과를 가져오는 버그를 포함하게 된다. 하지만 read() 함수의 실행 과정에서 오류가 발생하여 -1이 반환된 후에 2행이 실행되어 error 변수 값이 증가될 가능성은 일반적인 테스트 상황에서 드문 예라고 할 수 있다. 이러한 종류의 버그에 의한 사후 지원 비용의 발생을 미연에 방지할 수 있는 방법은 단위 검사 공정에서 가능한 모든 코드 경로를 검사해서 모든 경우에 올바른 결과가 나오는지 확인하는 것이다. 그러나 코드 경로는 조합되는 것이므로에 위 예에서 error 변수를 조건 블록 이전에 초기화한다면 에러가 발생하지 않는 좀 더 깔끔한 코드를 만들 수 있다.

다음 코드에는 어떠한 검토 방법으로도 찾아낼 수 있는 명백한 오류[05]가 포함되어 있다. 즉, 간단한 것을 복잡하게 만드는 것이 얼마나 쉬운지를 보여준다.

```
if (connect(s, &sa, &sa_len) == -1)
  error++;
else
  error = 0;

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

위의 예로부터 테스트해 볼 수 있는 네 가지 경우의 코드 경로를 만들 수 있다.

1. lines 1-2-5-6
2. lines 1-2-5-8
3. lines 1-4-5-6
4. lines 1-4-5-8

조합 가능한 모든 코드 경로를 테스트해본다는 것은 일반적으로 불가능하다. 왜냐하면 몇십 줄의 코드로 구성된 작은 함수의 경우에도 가능한 코드 경로는 수백 가지 이상이 될 수 있기 때문이다. 또한 단위 검사가 프로그램이나 모듈 전체의 코드를 연속적으로 모두 실행해 보는 방법 등을 사용해서 코드를 구성하는 모든 행을 확인할 수 있다고 확신하는 것만으로는 충분하지 않다. 이러한 종류의 커버리지 분석은 모든 소프트웨어 엔지니어가 사용할 수 있는 것이 아니며, 이것이 바로 QA[Quality Assurance 06]다.

회귀 테스트

버그를 수정하는 것만으로는 충분하지 않다. "문서 상으로 자세하게 살펴보는 것으로도 버그를 찾아낼 수 있다."는 말은 종종 오류를 확실하게 찾아낼 수 있는 테스트 도구를 만드는 것이 어렵기 때문에 변명처럼 사용되기도 한다. 0으로 숫자를 나누는 경우처럼 코드를 확인하는 것만으로도 쉽게 찾아낼 수 있는 명확한 버그가 많이 존재하지만 '무엇을 수정해야 하는지'를 찾아내기 위해서는 프로그래머의 의도를 정확하게 파악하기 위해서 주변의 코드를 함께 살펴 보아야만 한다. 이러한 종류의 분석은 수정이 이루어질 때 함께 문서화되거나 소스 코드에 주석의 형태로 포함되어야 한다.

그러나 문서 상의 검토를 통해서도 버그를 찾아낼 수 없는 경우가 좀 더 일반적이며, 버그 수정 또한 문제의 원인이 된 곳이 아닌 다른 코드 부분에 이루어지기도 한다. 이러한 경우에는 잘못된 코드 경로와 수정된 코드 부분을 검사해볼 수 있는 새로운 단위 검사를 만들어서 테스트해보아야 한다. 수정된 프로그램이나 모듈에 대한 검토와 확인이 끝난 뒤에는, 이후에 있을지도 모를 또 다른 수정에 의해서 이전의 버그와 유사한 종류의 버그가 재발되었을 때, QA를 통해서 고객들이 발견하기 이전에 문제점을 찾아내기 위해서 새로운 단위 검사 자체에 대한 검토와 확인이 이루어져야만 한다.

오픈 소스 소프트웨어 공학

오픈 소스 프로젝트에는 소프트웨어 공학 공정 중에서 하나의 공정만이 포함될 수도 있고 몇 개의 공정이 포함될 수도 있다. BSD와 BIND, sendmail의 상용 버전은 표준적인 소프트웨어 공학 공정이 적용된 사례라고 할 수 있다. 그러나 이 제품들의 개발이 처음부터 소프트웨어 공학으로 시작된 것은 아니었다. 완숙한 소프트웨어 공학 공정은 예산에 대한 계획이 필요한 관련 자료가 매우 드물고 실제적인 투자가 이루어져야 하는 작업이다.

그러나 오픈 소스 프로젝트에서 좀 더 일반적인 경우는 단순히 자신의 작업에 재미를 느끼고 자신이 만든 프로그램이 좀 더 널리 사용되기를 바라는 마음으로 배포상의 제한을 두지 않고 이를 무료로 배포하는 경우다. 이런 부류의 사람들은 코드 커버리지 분석기나 바운드 확인 인터프리터bounds-checking interpreter, 메모리 통합 검증기memory integrity verifier 등과 같은 소위 '상용 차원의' 개발 도구를 갖지 않는 경우가 우세하다. 또한 그들이 흥미를 느끼는 일차적인 것은 코딩과 패키징 그리고 오픈 소스의 전파에 있지, QA나 MRD에 있지 않으며 조급한 제품 출하시기에 얽매여 있지도 않다.

다시 한번 소프트웨어 공학 공정을 살펴보면서 애정과 노력 이외에는 별도의 기금이 지원되지 않는 오픈 소스 프로젝트에서 어떠한 대체 상황이 일어나는가를 알아보도록 하자.

요구 사항 분석

오픈 소스 개발자는 그들이 필요로 하거나 갖고 싶은 도구를 스스로 만드는 경향이 있다. 때때로 이러한 일은 그들의 직업상 필요에 의해서 만들어지곤 하는데 소프트웨어 공학보다는 시스템 관리와 같은 일에 종사하는 사람들에 의해서 만들어지는 경우가 많다. 이러한 과정이 몇 번 반복된 뒤에는 소프트웨어의 크기가 상당히 커지고 스스로의 생명력을 갖게 되어 인터넷 등을 통해서 tarball[07]의 형태로 배포되기에 이른다. 그리고 다른 사용자가 소프트웨어에 관심을 갖게 되어 질문을 하거나 스스로 여러 가지 기능을 직접 구현해서 보내게 된다.

오픈 소스 프로젝트에서 MRD가 이루어지는 곳은 대부분 메일링 리스트나 뉴스그룹이며 사용자와 개발자가 함께 직접 의견을 교환한다. 합의consensus는 개발자가 기억하거나 동의하는 것으로 이루어지며, 합의에 이르지 못하면 자신이 원하는 독립된 버전을 만들기 위해서 코드가 분리되는 결과를 가져온다. 오픈 소스에서도 MRD에 해당하는 부분을 키워나갈 수 있지만 의견 대립 때문에 합의점을 찾을 수

없는 경우가 종종 있고 합의를 도출하려는 시도도 이루어지지 않을 때가 있다.

시스템 디자인

일반적으로 오픈 소스 프로젝트에서는 시스템 디자인 단계가 없다고 보는 것이 타당할 듯 싶다. 오픈 소스에서의 시스템 디자인은 함축된 형태로 존재하다가 어느 날 갑자기 완성된 형태가 나타나든지, 아니면 소프트웨어처럼 점진적으로 발전되는 형태를 갖고 있다. 일반적으로 2 또는 3 버전에 이르게 되면 문서화되지 않았다고 하더라도 실제적인 시스템 디자인을 갖추게 된다.

바로 이러한 점이 오픈 소스가 소프트웨어 공학의 일반적인 기준으로부터 출발한 다른 소프트웨어에 비해서 신뢰성이 떨어진다는 평가를 받는 원인이 된다. 공식적인 MRD나 QA의 부족을 훌륭한 프로그래머나 사용자를 통해서 보상받을 수 있더라도, 문서화된 시스템 디자인이 빠져 있는 프로젝트의 품질은 자체적인 한계를 가질 수밖에 없다. 문서화되지만 않았을 뿐 누군가의 머릿속에 디자인이 구상되어 있다는 것은 아무런 변명거리가 될 수 없다.

세부 디자인

기금을 사용할 수 없는 오픈 소스에서 또 하나의 손실은 세부 디자인이다. 세부 디자인 문서인 DDD[Detailed Design Document]를 기꺼이 작성하려는 사람이 있지만, 이러한 경우에도 그들의 직업과 관련해서 근무 시간 중에 DDD를 작성하려고 할 뿐이며, 별도의 시간에 세부적인 DDD를 만들려고 하는 것은 아니다. 따라서 "이 작업에는 파서가 필요하므로 나중에 하나 만들어야 할 것 같다."라고 판단하는 경우처럼 세부 디자인 단계가 독립적인 공정이 되지 않고 구현 단계에서 필요한 요소를 만들면서 생겨나는 부수적인 것이 되어 버린다.

API를 전역 심볼의 형태로 헤더 파일 안에 기술하거나 매뉴얼의 형태로 문서화하는 작업은 선택적인 것이며, API를 공개하거나 이를 프로젝트 외부에서도 사용하

려는 의도가 없다면 세부 디자인 단계에서 이러한 종류의 문서화 작업은 이루어지지 않을 것이다.

그러나 세부 디자인 단계가 생략되거나 소홀해지는 것은 유감스러운 일이다. 왜냐하면 재사용될 수 있는 유용한 많은 코드가 이러한 이유로 인해서 사라져버리기 때문이다. 심지어 재사용되지 않거나 프로젝트에 크게 필요하지 않은 모듈이기 때문에 그 모듈의 API에 대한 설명이 필요하지 않다고 할지라도 API의 역할과 호출 방법을 설명하는 매뉴얼 작업은 반드시 이루어져야 한다. 이러한 문서는 코드를 향상시키려는 다른 사람에게 많은 도움이 될 수 있는데, 그 이유는 바로 이러한 문서를 읽고 이해하는 것으로부터 모든 작업이 시작될 수 있기 때문이다.

구현

구현은 매우 흥미있는 부분이다. 구현은 프로그래머가 가장 좋아하는 부분이며 졸음이 올 때까지 오랫동안 그들을 집중하게 하는 것이기도 하다. 코드를 작성할 기회는 거의 모든 오픈 소스 소프트웨어 개발에 대한 노력의 최우선적인 동기를 제공한다. 만약 소프트웨어 공학 측면의 다른 요소를 모두 배제하고 구현 단계만을 고려한다면 여기에는 많은 표현의 자유가 있다.

오픈 소스 프로젝트는 많은 프로그래머가 새로운 스타일을 시험할 수 있는 곳이다. 여기에는 들여쓰기indentation나 변수의 이름짓는 법 또는 메모리 비용이나 CPU 사이클을 줄이는 방법 등이 포함된다. 멋진 작품들이 tarball의 형태로 도처에 깔려 있고 프로그래머는 그들만의 새로운 스타일을 적용하고 동작하게 하려고 애쓴다.

기금이 제공되지 않는 오픈 소스의 구현에는 원하는 만큼의 엄격함과 지속성이 확보될 수 있다. 사용자는 코드가 가진 기능을 시험해볼 뿐이며, 개발자가 구현 과정에서 스타일을 바꿨는지 아니지는 관여하지 않는다.

물론 개발자는 이러한 변화에 신경을 쓰거나 신경을 쓰려고 할 것이다. 이러한 상황에서는 래리 월이 예전에 말했던 '예술적 표현으로서의 프로그래밍'[08]이라는 언급이 매우 적절한 것이라 할 수 있다.

오픈 소스 프로젝트의 구현 공정이 가진 가장 중요한 차이는 검토 과정이 비공식적으로 진행된다는 점이다. 코드가 완성되기 전에 멘토링이나 등위 검사 등이 사용되는 일은 일반적으로 매우 드문 일이며, 단위 검사나 회귀 검사와 같은 것도 존재하지 않는다.

통합

오픈 소스 프로젝트에서의 통합 과정은 일반적으로 온라인 매뉴얼을 작성하고 개발자들이 접근할 수 있는 모든 시스템에서 올바르게 빌드되는지를 확인하고, 구현 과정에서 포함된 부수적인 코드를 없애기 위해 서 Makefile을 정리하고 README 파일을 만들고 tarball을 만들어서 익명 FTP 사이트에 올린 뒤에 프로그램에 관심을 가질 만한 사람이 알 수 있는 메일링 리스트나 뉴스 그룹에 소식을 포스팅하는 것이다.

comp.sources.unix 뉴스 그룹은 1998년도에 롭 브라운Rob Broun에 의해서 다시 불붙기 시작했으며, 여기에 새로운 오픈 소스 소프트웨어 패키지나 업데이트 정보를 올리는 것은 매우 좋은 방법이다. 이곳은 저장소repository나 아카이브archive의 역할을 하기도 한다.

시스템 차원의 검사가 없다는 것이 별다른 문제가 되지는 않는다. 하지만 그렇게 되면 시스템 차원의 검사 계획이나 단위 검사는 존재하지 않게 된다. 오픈 소스의 경우에는 전체적으로 검사에 대한 비중을 별로 두지 않는 편이다(Perl이나 PostgreSQL 같은 예외도 존재한다). 그러나 제품을 발표하기 전 검사 공정이 결여되었다는 것은 다음에서 설명될 이유로 인해서 오픈 소스의 단점이 되지는 않는다.

필드 테스트

기금 없이 진행되는 오픈 소스 소프트웨어 개발 공정은 미항공우주국NASA의 우주 로봇 테스트 공정을 비교의 대상으로 삼지 않는다면 산업계에서 가장 좋은 시스템 차원의 테스트 환경을 가진다고 볼 수 있다. 왜냐하면 사용자는 그들이 돈을 지불할 필요가 없을 때 프로그램에 더욱 친근하게 다가서게 되는 경향이 있으며, 개발자 자신을 포함하는 파워 유저는 그들이 실행하고 있는 프로그램의 소스 코드를 읽고 고칠 수 있을 경우에 좀 더 많은 도움을 줄 수 있기 때문이다.

필드 테스트의 본질은 엄격함이 결여되어 있다는 점이다. 소프트웨어 공학이 필드 테스터에게 요구하는 것은 시스템이 디자인되고 빌드될 때 본질적으로 예측될 수 없었던 사용 습관에 대한 것이다. 다시 말하면 실제 사용자의 실제 사용 경험인 것이다. 오픈 소스 프로젝트는 이러한 부분에 있어서 오히려 강점을 가진다.

오픈 소스 프로젝트로부터 향유할 수 있는 또 하나의 이점은 단순히 프로그램을 실행하는 것보다 소스 코드를 읽는 방법으로 버그를 찾아내려는 많은 프로그래머에 의한 등위 검토가 가능하다는 점이다. 어떤 사람은 보안 상의 허점을 찾아내려고 노력하며 발견된 결함의 일부를 다른 사람에게 알리지 않기도 한다. 그러나 이러한 부분적인 위험이 소스 코드를 읽는 불특정 다수의 사람에 의해서 수행될 수 있는 전체적인 이점을 감소시키지는 못한다. 이러한 불특정 다수가 기존의 어떠한 관리자나 멘토가 하지 못했던 것을 가능하게 할 수 있도록 오픈 소스 개발자를 지원할 수 있는 것이다.

사후 지원

오픈 소스에서 사용자들이 버그를 찾아냈을 때는 “이런, 미안합니다!” 등의 말을 들을 수 있으며 개발자에게 버그 패치를 보내줄 경우에도 “미안합니다. 그리고 고맙습니다.” 정도의 말을 들을 수 있을 뿐이다. 만약 개발자가 버그 패치에 대해서 별다른 신경을 쓰고 싶지 않은 경우라면 “저는 별문제 없이 실행되는데요.” 등의

말을 하게 된다. 오픈 소스에서 흔하게 일어나는 이러한 상황이 혼란스럽게 느껴진다면 그럴 수도 있다. 책임 있는 사후 지원이 제공되지 않는다는 점이 오픈 소스 프로그램에 다가서려는 사람의 숫자를 줄이는 원인이 될 수도 있겠지만, 이러한 점이 컨설턴트나 소프트웨어 배포업체로 하여금 지원 서비스를 판매할 수 있는 기회와 오픈 소스 프로그램을 상용 버전으로 발전시킬 수 있는 기회를 제공할 수도 있다.

유닉스 제조업체가 고객에게 그들의 시스템에 오픈 소스 소프트웨어를 함께 탑재해줄 것을 강하게 요구받았을 때 나온 그들의 첫 번째 반응은 "함께 제공은 하지만 오픈 소스에 대한 지원은 하지 않는다."는 것이다. 시그너스 솔루션과 같은 기업의 성공은 이러한 정책을 재검토해서 틈새 시장을 공략한 점에서 찾을 수 있다. 하지만 오픈 소스와 기존의 기업 사이에는 아직도 상당한 문화적인 차이가 존재한다. 유닉스 제조업체를 포함한 전형적인 소프트웨어 제조 회사는 오픈 소스 메커니즘을 통해서 프로그램에 불특정 다수가 만든 검증되지 않은 변화가 일어날 경우에는 지원 사업의 판매 비용에 대한 계획이나 예산을 세울 수는 없다.

때때로 이러한 문제에 대한 해답은 소프트웨어가 단위 검사와 시스템 검사, 코드 커버리지 분석 등을 포함한 일반적인 QA 공정을 거칠 수 있도록 만드는 것이다. 이러한 작업에는 어떠한 기능을 검사해야 하는지를 찾아내기 위해서 리버스 엔지니어링을 이용해서 MRD와 DDD를 만든 후에 QA 공정에 사용할 수 있다. 또 다른 방법은 제품에 대한 지원 계약을 체결할 때 납품된 제품에 대한 '완벽한 보증'이 아니라 '최상의 지원'이라는 하향된 용어를 사용하는 것이다. 궁극적으로 소프트웨어 지원 시장은 오픈 소스에 참여하는 불특정 다수로부터 힘을 이끌어 낼 수 있는 사람들로 채워질 것이다. 왜냐하면 이러한 불특정 다수 중의 많은 사람은 좋은 소프트웨어를 만들 수 있는 사람이고, 오픈 소스의 문화는 사용자가 실제로 원하는 기능을 만들어내는 데 있어서 매우 효과적이기 때문이다(리눅스와 윈도우의 경우를 비교해 보면 이를 쉽게 알 수 있을 것이다).

결론

공학은 오래된 기술 분야며 소프트웨어를 만들건 하드웨어를 만들건 아니면 기차 철로를 건설하든 간에 근본적으로 다음과 같은 동일한 구성 요소를 가진다.

요구 사항과 그것을 필요로 하는 수요자를 명확하게 파악한다.

요구 사항을 충족할 수 있는 해결 방법을 디자인한다.

디자인을 모듈화하고 구현 계획을 세운다.

실제로 구현하고 검사한 뒤에 완성품을 납품하고 여기에 대한 사후 지원을 제공한다.

어떤 분야에서는 특정한 공정에 특히 중점을 두기도 한다. 예를 들어 철로 시공 업체에서는 MRD나 구현 공정, 그리고 사후 지원에 대해서 특별히 많은 관심을 기울이지 않는다. 그러나 그들은 QA와 SDD(System Design Document), DDD(Detailed Design Document)에 대해서는 각별한 주의를 기울인다.

프로그래머가 소프트웨어 엔지니어로 발돋움하기 위해서는 소프트웨어 공학도 하나의 독립된 분야라는 인식을 갖고 이 분야에 들어오기 위해서 소프트웨어 공학적인 새로운 사고 방식과 많은 노력이 요구된다는 것을 깨닫는 것이 필요하다. 오픈 소스 개발자는 종종 프로그래밍과 소프트웨어 공학과의 차이점을 깨닫고 이를 실무에 적용하기 전부터 성공을 거두기도 했는데, 이는 단순히 오픈 소스 프로젝트가 공학 측면의 결여로 인한 어려움을 오래전부터 감수해왔던 관성에 의한 것이다.

나는 이 글을 통해서 소프트웨어 공학에 대한 매우 단편적인 개괄을 소개했으며, 좀 더 많은 오픈 소스 프로그래머에게 이 분야로 투신하기 위한 관심과 동기를 제공했기를 희망한다. 미래는 과거와 현재까지 성취되어 온 가장 좋은 것의 결합을 통해서 이루어진다는 사실을 잊지 말기 바란다.

소프트웨어 공학은 단순히 계산척이나 주머니 보호구[09]와 같은 것을 만들기 위한 것이 아니다. 고품질의 시스템을 만들어내기 위해서 많은 검증된 기술이 이용되고 있는 폭넓은 분야며, 특히 일반적인 오픈 소스 프로젝트처럼 '한 명의 특출난 프로그래머'에게 매달리지 않아도 고품질의 시스템을 만들어낼 수 있는 길이기도 하다.

---

01 역자주_소프트웨어 공학(Software engineering)은 1967년부터 등장하기 시작한 개념으로 소프트웨어의 개발과 유지 관리 등의 작업에 공학적인 접근 방법을 도입해서 기존에 존재했던 문제점을 해결하기 위한 방법론적인 접근 방식이다. 우리나라에서는 1990년대 중반부터 본격적으로 도입되고 있는 신생 분야 중의 하나이며, 추천할만한 개론서로 이안 솜머빌(Ian Sommervile)의 『Software Engineering』 (Addison-Wesley, 5th Edition, 1995)이 있다. 소프트웨어 공학론에 관련된 기념비적인 저작인 프레더릭 브룩스(Frederick Brooks)의 『The MythicalManMonth』 (Addison-Wesley, Anniversary Edition, 1995)는 필독을 권할만한 의미깊은 책이다.

02 역자주_소프트웨어 공학 공정의 진행 단계는 순차적인 과정을 따르는 형식을 갖기 때문에 '폭포수 모델(Waterfall Model)'이라고 부르기도 한다. 이 모델은 1970년에 로이스(W. W. Royce)가 발표한 『Managing the Development of Large Software Systems: Concepts and Techniques』라는 논문을 통해서 소개된 이후에 가장 기본적인 소프트웨어 개발 공정 모델로 사용되고 있다.

03 역자주_PERT 차트는 Program Evaluation and Review Technique의 약자로 프로젝트를 진행하는 데 필요한 단위 작업과 소요 일정 등을 노드와 간선으로 연결한 것이다. PERT 차트는 작업 사이의 우선순위와 의존 관계, 프로젝트에 필요한 최소 일정 등을 산출하는 데 사용할 수 있다. GANT 차트는 헨리 간트(Henry L. Gantt)에 의해서 제안된 것으로 프로젝트의 진행 계획을 시간과 목적의 두 가지 항목을 갖는 표로 구성한 것이다. GANT 차트는 일정 계획, 진행 관리, 실적 차트 등에 이용될 수 있다. 프로젝트 차트를 생성해줄 수 있는 대표적인 프로그램으로는 마이크로소프트의 Project가 있으며 리눅스 환경에서는 Projector라는 프로그램이 있다. 해당 정보 사이트는 http://www.microsoft.com/korea/office/project/default.htm 과 http://www.iedu.com/project이다.

04 역자주_리눅스 환경에서 실험해볼 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 참

고하면 실행된 코드 경로와 조건 분기가 일어난 곳 등의 실행 정보를 참고할 수 있다. gcov에 대한 좀 더 자세한 사항은 http://gnu.kldp.org/opensources/gcov_toc.ko.html 문서를 통해서 참고할 수 있다.

```
[chsong@slug]$ 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
[chsong@slug]$ gcc -fprofile-arcs -ftest-coverage test.c
[chsong@slug]$./a.out
[chsong@slug]$ gcov -b test.c
[chsong@slug]$ cat test.c.gcov
```

05 역자주_예를 들어 네 번째 코드 경로인 1-2-5-8의 경우에는 발생한 에러의 개수가 모두 1개임에도 불구하고 8행에 의해서 에러의 수가 0으로 변경되는 오류를 갖고 있다. error 변수는 프로그램이 실행되는 과정에서 시스템 콜에 의해서 발생한 에러의 수를 저장하는 변수다.

06 역자주_QA는 개발 공정 자체를 모니터링하고 개선하는 작업과 정해진 표준을 지켰는지와 검사 공정 중에 문제가 발생하지 않았는지와 같은 소프트웨어 개발 공정 전체에 대한 사항을 모두 포괄하는 분야다. QA를 통해서 잘못된 점을 사전에 감지해서 오류를 방지하거나 수정할 수 있다.

07 역자주_tar로 묶여 있는 파일을 지칭하는 말로 빈번하게 사용된다.

08 역자주_예술적 표현(artistic expression)이란 1997년 8월 20일에 열렸던 제2회 Perl 콘퍼런스에서 행해진 그의 기조연설에서 대중적으로 소개된 말로 http://kiev.wall.org/~larry/keynote/keynote.html 문서의 Artistic Beliefs 부분을 통해서 자세한 내용을 참고할 수 있다. http://g2.songline.com:8080/ramgen/ntserver1/perl-com/larry-tpc2.rm를 통해서 1998년 8월 25일에 열렸던 제2회 Perl 콘퍼런스에서 행해진 래리 월의 기조연설을 육성으로 직접들을 수 있는데, 그의 번뜩이는 기지와 철학적인 내면을 살펴볼 수 있는 매우 추천할만한 자료다. 이 연설의 내용은 http://www.perl.com/print/1998/08/show/onion.html에서 문서 자료로 참고할 수도 있는데 이 연설의 내용을 다듬어서 출판한 것이 바로 이

책의 9번 째 글에 해당하는‘근면, 인내, 그리고 겸손’이며, 1999년에 열렸던 제3회 Perl 콘퍼런스에서는 좀더 발전된 사색의 결과를 엿볼 수 있는 기조연설이 행해지기도 했다. 그의 세 번째 기조연설은 http://www.perl.com/1999/08/onion/talk.html를 통해서 참고할 수 있다.

09 역자주_계산척(slide rule)과 주머니 보호구(pocket protector)는 소프트웨어 공학이 기계 공학적 실물을 만들어내는 데만 사용되는 것이 아님을 설명하기 위해서 예를 든 것이다. 계산척은 계산할 수 있는 자이며 주머니 보호구는 연필 등의 뾰족한 필기구에 의해서 주머니가 손상되지 않도록 만든 작은 휴대용 주머니다.

7 | 첨단의 리눅스

리누스 토발스Linus 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 정복world 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 실행한다write 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 썬Sun이 "한 번 작성하여 모든 곳에서 실행한다."는 아이디어를 발명해낸 것은 아니다. 이식성Portability은 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 팜파일럿부터 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 버클리 유닉스의 소스 코드를 가지고 시작했으며 그 커널은 버클리 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 포트port 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, 아미가Amiga 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 포크fork(분리되어 나온 것)라고 할 수 있다.

한편으로는 이 첫 번째 68K 리눅스가 이식성이 뛰어난 리눅스를 만드는 데 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 시스템을 만들 수 있는가였다. 여기에 대해 인정받던 학파가 있었는데 그들의 주장은 마이크로커널 스타일의 아키텍처를 사용해야 한다는 것이었다.

리눅스 커널과 같은 모노리딕monolithic 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 몇가지 저수준 입/출력이 이에 속한다. 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 방식보다 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 마이크로 커널의 최적화 작업에 사용되었다. 재미있는 사실은 여러분이 실제로 논문을 읽어 보면 연구자들이 마이크로커널

에 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 갖지 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.

이식Porting, 포팅이라는 문제는 몇 가지 가정을 통해 단순화할 수 있다. 예를 들어 CPU에 페이징을 해야 한다면 CPU가 사용하는 가상 메모리를 매핑할 수 있는 변환 참조 버퍼Translation 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 부분

을 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 3단계로만 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 오늘날의 컴퓨터 아키텍처에 있어서 존재하는 현실에 최상의 이론을 결부시키기 위해 중간자적 입장을 고수하려 노력해왔다.

커널 영역과 사용자 영역

리눅스 커널과 같은 모노리딕 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 꼼짝 없이 그 인터페이스에 갇히게 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 클론clone 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 성능 이 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 작게 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 GNU Public License로 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 머세드Merced가 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 모듈화module되어 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 라이저Hans Reiser는 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 런타임run-time 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. 유래한 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 헛점을 남기고 있으며, 이는 부분적으로 GPL이 모듈 인터페이스와 같은 것에 대하여 명확하게 하지 않음이기도 하다.

만약 누군가 단지 GPL을 우회하기 위해 엑스포트된 심볼exported symbol을 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 이룰 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 이식성의 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 갱신 사항을 미리 생각하지 않는다. 커널에서 중요한 혁신은 그리 많지 않

다. 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 클러스터링을 하게 되면 여러분은 갑자기 스케줄러에게 특정 그룹의 프로세스에 대하여 집단 스케줄링gang scheduling과 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 시스템embedded system이 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 인퍼노Inferno(루슨트의 내장 시스템용 운영체제)가 내장 장치에서 성공할 것이라고 생각하지 않는다. 그들은 무어의 법칙Moore's Law이 가진 중요성을 망각하였다. 맨 처음에는 특정 내장 장치에 최적화된 시스템을 설계하는 것이 그럴듯하게 들릴지는 모르나 작동 가능한 설계를 가질 때쯤에는 무어의 법칙에 의거하여 좀더 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 프로세싱SMP은 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. 현

재 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. 한편 웹 서비스는 내게 위험한 영역이다. 왜냐하면 리눅스를 웹 서비스 플랫폼으로 사용하는 사람들의 피드백을 많이 받음으로써 단지 웹 서비스에 대해서만 최적화하는 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 에뮬레이터WINE가 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.