CHAPTER 1

Introduction: How UNIX Gave Birth to Linux, and a New Software Paradigm

Lawyers and businesspeople who are first learning about open source tend to think of it as an entirely new paradigm, or a disruptive technology. But open source is easier to understand within its historical context. It is true that open source software licensing is the biggest sea change in technology licensing since software licensing began. But the more things change, the more they stay the same. This chapter outlines the historical background for the free software movement and the later open source movement, and explains why and how they arose.

In the Beginning Was the Word, and the Word Was UNIX

The term "open source" refers primarily to a type of outbound licensing paradigm, but also to a method of software development. Although media attention to both of these aspects of open source has burgeoned in the last decade, both the licensing paradigm and the development method have been in use ever since modern software was developed.

Although there are many software applications and utilities licensed under open source schemes, the "killer app" of open source is the Linux

operating system.[1] Understanding the free software movement of the 1990s without understanding UNIX is a little like trying to understand Martin Luther without knowing who the pope is—you may learn the doctrines of Protestantism, but they will seem arbitrary if you do not know their historical context. The philosophical tenets of open source can seem arbitrary out of context, and many lawyers and businesspeople struggle with them for this reason: Those who grew up in the age of Windows do not know much about an operating system, like UNIX, that they have never used.

The company that came up with the first modern computer operating system was not a software company but a telephone company. UNIX was developed by AT&T Bell Laboratories back when AT&T was a much-feared corporate monopoly. As a result, the company was operating under a consent decree from the Department of Justice that required AT&T not to engage in commercial activities outside the field of telephone service. AT&T had enormous research and development resources, and boasted among its ranks some of the best and brightest computer engineers of the day. AT&T set its engineers loose to develop technology within the corporate context of a not-for-profit subsidiary called AT&T Bell Laboratories.

In the 1970s, two scientists at Bell Labs, Ken Thompson and Dennis Ritchie, not only came up with UNIX, but invented a computer programming language in which to write it. That language was called C. The origin of the name UNIX may be apocryphal, but it was allegedly a pun on the word "eunuchs"; it is a quasi-acronym for Uniplexed Information and Computing System, and a successor to the earlier Multiplexed Information and Computing Service (MULTICS), but with more focused functionality. UNIX and C were tremendously innovative. UNIX was written to operate the computers of the day: large mainframes whose use was limited to huge corporations, such as banks and utilities, government, and academia. C was an extraordinarily flexible and powerful programming language. Many of the languages today, C++ and Java, for instance, are heavily based on the syntax of C.

[1]This is more completely called the GNU/Linux operating system, a distinction to be explained later in this chapter.

Because of the consent decree, AT&T was not allowed to exploit UNIX as a commercial product. So it licensed copies of UNIX to universities and others all over the world for one dollar. It soon became common practice for computer scientists to share their improvements and innovations for UNIX freely—no one tried to exploit the modifications, because there was no serious market for the product. Computers were still in use by relatively few organizations, all software was custom-written and had to be configured and installed individually, and there was no consumer computer industry.

Eventually, the consent decree was lifted. By this time, UNIX was in wide use by academics who were accustomed to treating it like a scholarly research project, not as a commercial product. UNIX was being used in at least two ways: to run computers to support other academic projects, such as statistical analyses and scientific calculations, and as a device for teaching students the nuts and bolts of operating system design. After the decree was lifted, AT&T started granting commercial licenses for UNIX, under original equipment manufacture (OEM)–type commercial licensing terms.[2] Once this happened, the many UNIX licensees no longer shared their modifications, which caused what is called *forking*: the development of many incompatible versions. Each vendor—IBM, Sun, and even Microsoft—developed its own "flavor" of UNIX licensed in object code form only.

The free software movement was a direct reaction to the privatization of UNIX. Computer scientists, particularly academics, thought operating systems needed to be freely available in source code form. Thus, "free software" refers to free availability of source code, not price. (As the Free Software Foundation pithily says, "Think free speech, not free beer.") This free availability was important because an operating system is a fundamental tool, and if it works improperly, slowly, or badly, all users suffer. Thus, free software became a political movement, based on a normative idea that the forking and inaccessibility of UNIX should never happen again. It is no accident that the computer science luminaries who

[2]This term is not used consistently in the industry; here I use it to mean a source code license allowing the OEM to make modifications but to distribute object code only, where the source code is designated a trade secret.

initiated the movement were men who started their careers using UNIX and struggled with the problems that its privatization engendered.

Along Comes Linux

Now, a few factors dovetailed to make Linux the vehicle of the free software movement.

In the late 1980s, computer science was undergoing a revolution. UNIX was no longer the most common operating system. This period saw the rise and fall of the first microcomputers: the Apple II, the TRS-80, and, of course, the personal computer. UNIX did not run on those boxes. There was no UNIX-like operating system for the new, cheap, Intel-type processors that were beginning to dominate the desktop market. These processors ran on DOS and later the Windows operating system, both products of Microsoft.

Several people tried to write smaller, more nimble operating systems that would be useful alternatives to UNIX. Such systems had to be compatible with UNIX programs, but free of the intellectual property rights of AT&T. Most notably, systems emerged based on the then-current UNIX interface specification. UNIX compatibility was essential so UNIX applications could run on those systems. One was a scholarly project called MINIX, written by Andrew Tanenbaum to help him teach operating systems and software at Vrije University in Amsterdam; it filled the academic void that the privatization of UNIX left behind. The other, to become far more famous, was Linux, the first version of which was written by a teenage computer programmer in Helsinki named Linus Torvalds. Torvalds released the first version of Linux in 1991.

Meanwhile, the GNU Project had developed into a major project to build a free alternative to UNIX. (GNU is a recursive acronym for "GNU's not UNIX.") By the early 1990s, this project had been in operation for many years. The mission of the GNU Project was to build an entire operating system, whereas Linux was only a kernel. An operating system includes not only a kernel, but development tools like compilers, debuggers, text editors, user interfaces, and administrative tools. The GNU Project was struggling with kernel development, and Linux arrived in time to provide the final piece of the puzzle. In tandem with the GNU Project, Richard Stallman of the GNU Project pioneered free software by developing the GNU

General Public License. The Free Software Foundation (FSF), a not-for-profit organization that supports the GNU Project, became the publisher and steward of that license. The FSF convinced Torvalds to make his kernel available under free software licensing terms, and the rest is history. The GNU/Linux operating system is what most people call Linux.

Finally, the last element of serendipity occurred: the economic recession of the early 2000s. A nosediving technology industry was desperate to cut costs and keep innovation alive, and lots of programmers were out of work and looking for something to do to keep embarrassing gaps out of their resumes. This was the perfect environment for open source to blossom. Blossom it did. Statistics on adoption of Linux are hard to come by, but most agree that it is gaining exponentially in popularity, particularly outside the United States.

Now, What Is Open Source?

To understand open source, you must understand what source code is and why access to it is important. Most computer users today use desktop computers with Windows operating systems. When you run a program on your desktop computer, such as a word processing program or an e-mail program, the file on your computer that contains the program is called, for instance, myprogram.exe. The "exe" stands for "executable." The file you are accessing is an executable file, or a file containing instructions that the computer can read and perform.

Programmers do not write executable code, they write source code. Programmers and corporate lawyers are a lot alike, in that they rarely write anything from scratch. When we write a contract, we rarely sit down to a blank page to begin. We use a model or form agreement, and add provisions from our libraries of provisions, which we have developed or collected over the years. Programmers work in exactly this way. If a programmer were writing a word processing program, he or she would need to include a way to save a file to a disk drive. But it would not be efficient—or even advisable—for the programmer to write it from scratch. Rather, he or she would use a prewritten component, or "library routine," to accomplish this. By using a prewritten routine, programmers make their coding more efficient, and they have more assurance that the

code will be free of bugs and interoperable with the platform on which the program will run.

Lawyers writing contracts reuse provisions informally and idiosyncratically, but in programming, this process is highly formal. If you are writing a program and want to use a routine to write a file to disk, for example, the development language you are using will include a library routine called something like "writefile." That routine will need some information, such as where the file will be written, the name of the file, the information to be written, and how many bytes will be required. The documentation for the "writefile" routine will specify what information needs to be communicated to the routine when it is executed. Lawyers should think of this as similar to conforming an arbitration provision in a contract with the rest of the agreement. Definitions of "parties" and "business days" should not conflict. When writing contracts, lawyers do this by making global changes to the text. Those who have never written a contract might consider instead the example of writing a slew of thank-you letters. The letter might say: "Dear_____. Thank you for the lovely_____." Part of the letter is the same every time, and other parts vary. The parts that are the same are dictated by custom and manners.

In programming, this reuse is done formally and systematically. Programmers write their program in a text processor (or development environment). The code looks like cryptic English, and comprises a series of instructions telling the computer's processor what to do. Any skilled programmer can read this code, which is called source code. But the computer cannot execute this code as it is written. Once programmers have all the code written, and have included references to the library routines they need in that code, they run a large, complex program called a compiler. The compiler translates the source code into object code—a set of binary instructions that the computer's processor can execute. A related program, called a linker, then links the resulting object code to the referenced library routines, making sure information will be passed properly between the components, and produces a program that can be executed by the computer: an "executable" file, which usually contains many object code files. It is important to understand that programmers do not necessarily need access to the source code for the library routines. They only need to know what information to send to the routines and what

information they will send back. The routines are usually "black boxes" to programmers: They do not need to know what is inside the box, just what goes in and what comes out.

To use another example that lawyers and businesspeople may recognize, a compiler is a lot like a redlining program. It runs in "batch" mode; in other words, once you enter some information (such as identifying the current and prior version and how you want the deletions to look) and press "go," the program runs without user interaction. The end product is a file that you do not edit. If you discover a mistake in the redline, you must go back to fix the document, then rerun the redlining program. Similarly, if there is a bug in a computer program, you do not edit the object or executable file. You must correct the source code and recompile the program. This is why access to source code is crucial. Without source code, you cannot fix errors, and must rely on the program vendor to do so.

And This Is Just the Beginning

Open source, then, is not exactly new. Many of its tenets and practices are almost as old as the software industry. However, open source has come to be a focus of legal discussion as a result of having transformed from a set of informal industry practices to a political movement, with attendant intellectual property licensing practices.

Open source software licensing is a very complex topic. Some legal issues related to it are quite thorny and undecided. Also, formulating best practices in open source development requires familiarity with a complex set of facts and industry practices as well as the political, business, and legal principles at work. Best practices in this area change quickly and sometimes unexpectedly. This book, therefore, is not intended to give you all the answers. Instead, it is intended to provide you with the background and tools to understand this area of law and develop your own conclusions and best practices, to better leverage opportunities and manage risk.