xpub
T4PP

UNIX and Scriptable workflows

Oct 2016

- ## Two Bits: The Cultural Significance of Free Software

Type	Book
Author	Christopher M. Kelty
Publisher	Duke University Press
Date	208
Date Added	10/24/2016, 11:07:55 AM
Modified	10/24/2016, 11:09:14 AM

Notes:

- # UNIX History

  M. Kelty, Christopher. *Two Bits: The Cultural Significance of Free Software*. Duke University Press, 208AD.

  pp.125-142 (142-159)

  ```
  pdftk Two\ Bits_\ The\ Cultural\ Significance\ of\ Fre\ -\ Christopher\ M.\ Kelty.pdf cat 142-159 output TowBit-HistoryUnix.pdf
  ```

Attachments

- TowBit-HistoryUnix.pdf

Portability in the business world meant something specific, however. Even if software could be made portable at a technical level—transferable between two different IBM machines—this was certainly no guarantee that it would be portable between customers. One company's accounting program, for example, may not suit another's practices. Portability was therefore hindered both by the diversity of machine architectures and by the diversity of business practices and organization. IBM and other manufacturers therefore saw no benefit to standardizing source code, as it could only provide an advantage to competitors.[15]

Portability was thus not simply a technical problem—the problem of running one program on multiple architectures—but also a kind of political-economic problem. The meaning of *product* was not always the same as the meaning of *hardware* or *software*, but was usually some combination of the two. At that early stage, the outlines of a contest over the meaning of *portable* or *shareable* source code are visible, both in the technical challenges of creating high-level languages and in the political-economic challenges that corporations faced in creating distinctive proprietary products.

The UNIX Time-Sharing System

Set against this backdrop, the invention, success, and proliferation of the UNIX operating system seems quite monstrous, an aberration of both academic and commercial practice that should have failed in both realms, instead of becoming the most widely used portable operating system in history and the very paradigm of an "operating system" in general. The story of UNIX demonstrates how portability became a reality and how the particular practice of sharing UNIX source code became a kind of de facto standard in its wake.

UNIX was first written in 1969 by Ken Thompson and Dennis Ritchie at Bell Telephone Labs in Murray Hill, New Jersey. UNIX was the dénouement of the MIT project Multics, which Bell Labs had funded in part and to which Ken Thompson had been assigned. Multics was one of the earliest complete time-sharing operating systems, a demonstration platform for a number of early innovations in time-sharing (multiple simultaneous users on one computer).[16] By 1968, Bell Labs had pulled its support—including Ken Thompson—from the project and placed him back in Murray Hill, where he and

Dennis Ritchie were stuck without a machine, without any money, and without a project. They were specialists in operating systems, languages, and machine architecture in a research group that had no funding or mandate to pursue these areas. Through the creative use of some discarded equipment, and in relative isolation from the rest of the lab, Thompson and Ritchie created, in the space of about two years, a complete operating system, a programming language called C, and a host of tools that are still in extremely wide use today. The name UNIX (briefly, UNICS) was, among other things, a puerile pun: a castrated Multics.

The absence of an economic or corporate mandate for Thompson's and Ritchie's creativity and labor was not unusual for Bell Labs; researchers were free to work on just about anything, so long as it possessed some kind of vague relation to the interests of AT&T. However, the lack of funding for a more powerful machine did restrict the kind of work Thompson and Ritchie could accomplish. In particular, it influenced the design of the system, which was oriented toward a super-slim control unit (a kernel) that governed the basic operation of the machine and an expandable suite of small, independent tools, each of which did one thing well and which could be strung together to accomplish more complex and powerful tasks.[17] With the help of Joseph Ossana, Douglas McIlroy, and others, Thompson and Ritchie eventually managed to agitate for a new PDP-11/20 based not on the technical merits of the UNIX operating system itself, but on its potential applications, in particular, those of the text-preparation group, who were interested in developing tools for formatting, typesetting, and printing, primarily for the purpose of creating patent applications, which was, for Bell Labs, and for AT&T more generally, obviously a laudable goal.[18]

UNIX was unique for many technical reasons, but also for a specific economic reason: it was never quite academic and never quite commercial. Martin Campbell-Kelly notes that UNIX was a "non-proprietary operating system of major significance."[19] Kelly's use of "non-proprietary" is not surprising, but it is incorrect. Although business-speak regularly opposed *open* to *proprietary* throughout the 1980s and early 1990s (and UNIX was definitely the former), Kelly's slip marks clearly the confusion between software ownership and software distribution that permeates both popular and academic understandings. UNIX was indeed proprietary—it was copyrighted and wholly owned by Bell Labs and in turn by Western Electric

and AT&T—but it was not exactly commercialized or marketed by them. Instead, AT&T allowed individuals and corporations to install UNIX and to create UNIX-like derivatives *for very low licensing fees*. Until about 1982, UNIX was licensed to academics very widely for a very small sum: usually royalty-free with a minimal service charge (from about $150 to $800).[20] The conditions of this license allowed researchers to do what they liked with the software so long as they kept it secret: they could not distribute or use it outside of their university labs (or use it to create any commercial product or process), nor publish any part of it. As a result, throughout the 1970s UNIX was developed both by Thompson and Ritchie inside Bell Labs and by users around the world in a relatively informal manner. Bell Labs followed such a liberal policy both because it was one of a small handful of industry-academic research and development centers and because AT&T was a government monopoly that provided phone service to the country and was therefore forbidden to directly enter the computer software market.[21]

Being on the border of business and academia meant that UNIX was, on the one hand, shielded from the demands of management and markets, allowing it to achieve the conceptual integrity that made it so appealing to designers and academics. On the other, it also meant that AT&T treated it as a potential product in the emerging software industry, which included new legal questions from a changing intellectual-property regime, novel forms of marketing and distribution, and new methods of developing, supporting, and distributing software.

Despite this borderline status, UNIX was a phenomenal success. The reasons why UNIX was so popular are manifold; it was widely admired aesthetically, for its size, and for its clever design and tools. But the fact that it spread so widely and quickly is testament also to the existing community of eager computer scientists and engineers (and a few amateurs) onto which it was bootstrapped, users for whom a powerful, flexible, low-cost, modifiable, and fast operating system was a revelation of sorts. It was an obvious alternative to the complex, poorly documented, buggy operating systems that routinely shipped standard with the machines that universities and research organizations purchased. "It worked," in other words.

A key feature of the popularity of UNIX was the inclusion of the source code. When Bell Labs licensed UNIX, they usually provided a tape that contained the documentation (i.e., documentation that

was part of the system, not a paper technical manual external to it), a binary version of the software, and the source code for the software. The practice of distributing the source code encouraged people to maintain it, extend it, document it—and to contribute those changes to Thompson and Ritchie as well. By doing so, users developed an interest in maintaining and supporting the project precisely because it gave them an opportunity and the tools to use their computer creatively and flexibly. Such a globally distributed community of users organized primarily by their interest in maintaining an operating system is a precursor to the recursive public, albeit confined to the world of computer scientists and researchers with access to still relatively expensive machines. As such, UNIX was not only a widely shared piece of quasi-commercial software (i.e., distributed in some form other than through a price-based retail market), but also the first to systematically include the source code as part of that distribution as well, thus appealing more to academics and engineers.[22]

Throughout the 1970s, the low licensing fees, the inclusion of the source code, and its conceptual integrity meant that UNIX was ported to a remarkable number of other machines. In many ways, academics found it just as appealing, if not more, to be involved in the creation and improvement of a cutting-edge system by licensing and porting the software themselves, rather than by having it provided to them, without the source code, by a company. Peter Salus, for instance, suggests that people experienced the lack of support from Bell Labs as a kind of spur to develop and share their own fixes. The means by which source code was shared, and the norms and practices of sharing, porting, forking, and modifying source code were developed in this period as part of the development of UNIX itself—the technical design of the system facilitates and in some cases mirrors the norms and practices of sharing that developed: operating systems and social systems.[23]

Sharing UNIX

Over the course of 1974–77 the spread and porting of UNIX was phenomenal for an operating system that had no formal system of distribution and no official support from the company that owned it, and that evolved in a piecemeal way through the contributions

of people from around the world. By 1975, a user's group had developed: USENIX.[24] UNIX had spread to Canada, Europe, Australia, and Japan, and a number of new tools and applications were being both independently circulated and, significantly, included in the frequent releases by Bell Labs itself. All during this time, AT&T's licensing department sought to find a balance between allowing this circulation and innovation to continue, and attempting to maintain trade-secret status for the software. UNIX was, by 1980, without a doubt the most widely and deeply understood trade secret in computing history.

The manner in which the circulation of and contribution to UNIX occurred is not well documented, but it includes both technical and pedagogical forms of sharing. On the technical side, distribution took a number of forms, both in resistance to AT&T's attempts to control it and facilitated by its unusually liberal licensing of the software. On the pedagogical side, UNIX quickly became a paradigmatic object for computer-science students precisely because it was a working operating system that included the source code and that was simple enough to explore in a semester or two.

In *A Quarter Century of UNIX* Salus provides a couple of key stories (from Ken Thompson and Lou Katz) about how exactly the technical sharing of UNIX worked, how sharing, porting, and forking can be distinguished, and how it was neither strictly legal nor deliberately illegal in this context. First, from Ken Thompson: "The first thing to realize is that the outside world ran on releases of UNIX (V4, V5, V6, V7) but we did not. Our view was a continuum. V5 was what we had at some point in time and was probably out of date simply by the activity required to put it in shape to export. After V6, I was preparing to go to Berkeley to teach for a year. I was putting together a system to take. Since it was almost a release, I made a diff with V6 [a tape containing only the differences between the last release and the one Ken was taking with him]. On the way to Berkeley I stopped by Urbana-Champaign to keep an eye on Greg Chesson. . . . I left the diff tape there and I told him that I wouldn't mind if it got around."[25]

The need for a magnetic tape to "get around" marks the difference between the 1970s and the present: the distribution of software involved both the material transport of media *and* the digital copying of information. The desire to distribute bug fixes (the "diff" tape) resonates with the future emergence of Free Software: the

fact that others had fixed problems and contributed them back to Thompson and Ritchie produced an obligation to see that the fixes were shared as widely as possible, so that they in turn might be ported to new machines. Bell Labs, on the other hand, would have seen this through the lens of software development, requiring a new release, contract renegotiation, and a new license fee for a new version. Thompson's notion of a "continuum," rather than a series of releases also marks the difference between the idea of an evolving common set of objects stewarded by multiple people in far-flung locales and the idea of a shrink-wrapped "productized" software package that was gaining ascendance as an economic commodity at the same time. When Thompson says "the outside world," he is referring not only to people outside of Bell Labs but to the way the world was seen from within Bell Labs by the lawyers and marketers who would create a new version. For the lawyers, the circulation of source code was a problem because it needed to be stabilized, not so much for commercial reasons as for legal ones—one license for one piece of software. Distributing updates, fixes, and especially new tools and additions written by people who were not employed by Bell Labs scrambled the legal clarity even while it strengthened the technical quality. Lou Katz makes this explicit.

> A large number of bug fixes was collected, and rather than issue them one at a time, a collection tape ("the 50 fixes") was put together by Ken [the same "diff tape," presumably]. Some of the fixes were quite important, though I don't remember any in particular. *I suspect that a significant fraction of the fixes were actually done by non-Bell people.* Ken tried to send it out, but the lawyers kept stalling and stalling and stalling. Finally, in complete disgust, someone "found a tape on Mountain Avenue" [the location of Bell Labs] which had the fixes. When the lawyers found out about it, they called every licensee and threatened them with dire consequences if they didn't destroy the tape, after trying to find out how they got the tape. I would guess that no one would actually tell them how they came by the tape (I didn't).[26]

Distributing the fixes involved not just a power struggle between the engineers and management, but was in fact clearly motivated by the fact that, as Katz says, "a significant fraction of the fixes were done by non-Bell people." This meant two things: first, that there was an obvious incentive to return the updated system to these

people and to others; second, that it was not obvious that AT&T actually owned or could claim rights over these fixes—or, if they did, they needed to cover their legal tracks, which perhaps in part explains the stalling and threatening of the lawyers, who may have been buying time to make a "legal" version, with the proper permissions.

The struggle should be seen not as one between the rebel forces of UNIX development and the evil empire of lawyers and managers, but as a struggle between two modes of stabilizing the object known as UNIX. For the lawyers, stability implied finding ways to make UNIX look like a product that would meet the existing legal framework and the peculiar demands of being a regulated monopoly unable to freely compete with other computer manufacturers; the ownership of bits and pieces, ideas and contributions had to be strictly accountable. For the programmers, stability came through redistributing the most up-to-date operating system and sharing all innovations with all users so that new innovations might also be portable. The lawyers saw urgency in making UNIX legally stable; the engineers saw urgency in making UNIX technically stable and compatible with itself, that is, to prevent the *forking* of UNIX, the death knell for portability. The tension between achieving legal stability of the object and promoting its technical portability and stability is one that has repeated throughout the life of UNIX and its derivatives—and that has ramifications in other areas as well.

The identity and boundaries of UNIX were thus intricately formed through its sharing and distribution. Sharing produced its own form of moral and technical order. Troubling questions emerged immediately: were the versions that had been fixed, extended, and expanded still UNIX, and hence still under the control of AT&T? Or were the differences great enough that something else (not-UNIX) was emerging? If a tape full of fixes, contributed by non-Bell employees, was circulated to people who had licensed UNIX, and those fixes changed the system, was it still UNIX? Was it still UNIX in a legal sense or in a technical sense or both? While these questions might seem relatively scholastic, the history of the development of UNIX suggests something far more interesting: just about every possible modification has been made, legally and technically, but the *concept* of UNIX has remained remarkably stable.

Technical portability accounts for only part of UNIX's success. As a pedagogical resource, UNIX quickly became an indispensable tool for academics around the world. As it was installed and improved, it was taught and learned. The fact that UNIX spread first to university computer-science departments, and not to businesses, government, or nongovernmental organizations, meant that it also became part of the core pedagogical practice of a generation of programmers and computer scientists; over the course of the 1970s and 1980s, UNIX came to exemplify the very concept of an operating system, especially time-shared, multi-user operating systems. Two stories describe the porting of UNIX from machines to minds and illustrate the practice as it developed and how it intersected with the technical and legal attempts to stabilize UNIX as an object: the story of John Lions's *Commentary on Unix 6th Edition* and the story of Andrew Tanenbaum's Minix.

The development of a pedagogical UNIX lent a new stability to the concept of UNIX as opposed to its stability as a body of source code or as a legal entity. The porting of UNIX was so successful that even in cases where a ported version of UNIX *shares none of the same source code as the original*, it is still considered UNIX. The monstrous and promiscuous nature of UNIX is most clear in the stories of Lions and Tanenbaum, especially when contrasted with the commercial, legal, and technical integrity of something like Microsoft Windows, which generally exists in only a small number of forms (NT, ME, XP, 95, 98, etc.), possessing carefully controlled source code, immured in legal protection, and distributed only through sales and service packs to customers or personal-computer manufacturers. While Windows is much more widely used than UNIX, it is far from having become a paradigmatic pedagogical object; its integrity is predominantly legal, not technical or pedagogical. Or, in pedagogical terms, Windows is to fish as UNIX is to fishing lessons.

Lions's *Commentary* is also known as "the most photocopied document in computer science." Lions was a researcher and senior lecturer at the University of New South Wales in the early 1970s; after reading the first paper by Ritchie and Thompson on UNIX, he convinced his colleagues to purchase a license from AT&T.[27] Lions, like many researchers, was impressed by the quality of the system, and he was, like all of the UNIX users of that period, intimately

familiar with the UNIX source code—a necessity in order to install, run, or repair it. Lions began using the system to teach his classes on operating systems, and in the course of doing so he produced a textbook of sorts, which consisted of the entire source code of UNIX version 6 (V6), along with elaborate, line-by-line commentary and explanation. The value of this textbook can hardly be underestimated. Access to machines and software that could be used to understand how a real system worked was very limited: "Real computers with real operating systems were locked up in machine rooms and committed to processing twenty four hours a day. UNIX changed that."[28] Berny Goodheart, in an appreciation of Lions's *Commentary*, reiterated this sense of the practical usefulness of the source code and commentary: "It is important to understand the significance of John's work at that time: for students studying computer science in the 1970s, complex issues such as process scheduling, security, synchronization, file systems and other concepts were beyond normal comprehension and were extremely difficult to teach—there simply wasn't anything available with enough accessibility for students to use as a case study. Instead a student's discipline in computer science was earned by punching holes in cards, collecting fan-fold paper printouts, and so on. Basically, a computer operating system in that era was considered to be a huge chunk of inaccessible proprietary code."[29]

Lions's commentary was a unique document in the world of computer science, containing a kind of key to learning about a central component of the computer, one that very few people would have had access to in the 1970s. It shows how UNIX was ported not only to machines (which were scarce) but also to the minds of young researchers and student programmers (which were plentiful). Several generations of both academic computer scientists and students who went on to work for computer or software corporations were trained on photocopies of UNIX source code, with a whiff of toner and illicit circulation: a distributed operating system in the textual sense.

Unfortunately, *Commentary* was also legally restricted in its distribution. AT&T and Western Electric, in hopes that they could maintain trade-secret status for UNIX, allowed only very limited circulation of the book. At first, Lions was given permission to distribute single copies only to people who already possessed a license for UNIX V6; later Bell Labs itself would distribute *Commentary*

briefly, but only to licensed users, and not for sale, distribution, or copying. Nonetheless, nearly everyone seems to have possessed a dog-eared, nth-generation copy. Peter Reintjes writes, "We soon came into possession of what looked like a fifth generation photocopy and someone who shall remain nameless spent all night in the copier room spawning a sixth, an act expressly forbidden by a carefully worded disclaimer on the first page. Four remarkable things were happening at the same time. One, we had discovered the first piece of software that would inspire rather than annoy us; two, we had acquired what amounted to a literary criticism of that computer software; three, we were making the single most significant advancement of our education in computer science by actually reading an entire operating system; and four, we were breaking the law."[30]

Thus, these generations of computer-science students and academics shared a secret—a trade secret become open secret. Every student who learned the essentials of the UNIX operating system from a photocopy of Lions's commentary, also learned about AT&T's attempt to control its legal distribution on the front cover of their textbook. The parallel development of photocopying has a nice resonance here; together with home cassette taping of music and the introduction of the video-cassette recorder, photocopying helped drive the changes to copyright law adopted in 1976.

Thirty years later, and long after the source code in it had been completely replaced, Lions's *Commentary* is still widely admired by geeks. Even though Free Software has come full circle in providing students with an actual operating system that can be legally studied, taught, copied, and implemented, the kind of "literary criticism" that Lions's work represents is still extremely rare; even reading obsolete code with clear commentary is one of the few ways to truly understand the design elements and clever implementations that made the UNIX operating system so different from its predecessors and even many of its successors, few, if any of which have been so successfully ported to the minds of so many students.

Lions's *Commentary* contributed to the creation of a worldwide community of people whose connection to each other was formed by a body of source code, both in its implemented form and in its textual, photocopied form. This nascent recursive public not only understood itself as belonging to a technical elite which was constituted by its creation, understanding, and promotion of a particular

technical tool, but also recognized itself as "breaking the law," a community constituted in opposition to forms of power that governed the circulation, distribution, modification, and creation of the very tools they were learning to make as part of their vocation. The material connection shared around the world by UNIX-loving geeks to their source code is not a mere technical experience, but a social and legal one as well.

Lions was not the only researcher to recognize that teaching the source code was the swiftest route to comprehension. The other story of the circulation of source code concerns Andrew Tanenbaum, a well-respected computer scientist and an author of standard textbooks on computer architecture, operating systems, and networking.[31] In the 1970s Tanenbaum had also used UNIX as a teaching tool in classes at the Vrije Universiteit, in Amsterdam. Because the source code was distributed with the binary code, he could have his students explore directly the implementations of the system, and he often used the source code and the Lions book in his classes. But, according to his *Operating Systems: Design and Implementation* (1987), "When AT&T released Version 7 [ca. 1979], it began to realize that UNIX was a valuable commercial product, so it issued Version 7 with a license that prohibited the source code from being studied in courses, in order to avoid endangering its status as a trade secret. Many universities complied by simply dropping the study of UNIX, and teaching only theory" (13). For Tanenbaum, this was an unacceptable alternative—but so, apparently, was continuing to break the law by teaching UNIX in his courses. And so he proceeded to create a completely new UNIX-like operating system that used not a single line of AT&T source code. He called his creation Minix. It was a stripped-down version intended to run on personal computers (IBM PCs), and to be distributed along with the textbook *Operating Systems*, published by Prentice Hall.[32]

Minix became as widely used in the 1980s as a teaching tool as Lions's source code had been in the 1970s. According to Tanenbaum, the Usenet group comp.os.minix had reached 40,000 members by the late 1980s, and he was receiving constant suggestions for changes and improvements to the operating system. His own commitment to teaching meant that he incorporated few of these suggestions, an effort to keep the system simple enough to be printed in a textbook and understood by undergraduates. Minix

was freely available as source code, and it was a fully functioning operating system, even a potential alternative to UNIX that would run on a personal computer. Here was a clear example of the conceptual integrity of UNIX being communicated to another generation of computer-science students: Tanenbaum's textbook is not called "UNIX Operating Systems"—it is called *Operating Systems*. The clear implication is that UNIX represented the clearest example of the principles that should guide the creation of any operating system: it was, for all intents and purposes, state of the art even twenty years after it was first conceived.

Minix was not commercial software, but nor was it Free Software. It was copyrighted and controlled by Tanenbaum's publisher, Prentice Hall. Because it used no AT&T source code, Minix was also legally independent, a legal object of its own. The fact that it was intended to be legally distinct from, yet conceptually true to UNIX is a clear indication of the kinds of tensions that govern the creation and sharing of source code. The ironic apotheosis of Minix as the pedagogical gold standard for studying UNIX came in 1991–92, when a young Linus Torvalds created a "fork" of Minix, also rewritten from scratch, that would go on to become the paradigmatic piece of Free Software: Linux. Tanenbaum's purpose for Minix was that it remain a pedagogically useful operating system—small, concise, and illustrative—whereas Torvalds wanted to extend and expand his version of Minix to take full advantage of the kinds of hardware being produced in the 1990s. Both, however, were committed to source-code visibility and sharing as the swiftest route to complete comprehension of operating-systems principles.

Forking UNIX

Tanenbaum's need to produce Minix was driven by a desire to share the source code of UNIX with students, a desire AT&T was manifestly uncomfortable with and which threatened the trade-secret status of their property. The fact that Minix might be called a fork of UNIX is a key aspect of the political economy of operating systems and social systems. *Forking* generally refers to the creation of new, modified source code from an original base of source code, resulting in two distinct programs with the same parent. Whereas the modification of an engine results only in a modified engine, the

modification of source code implies differentiation *and* reproduction, because of the ease with which it can be copied.

How could Minix—a complete rewrite—still be considered the same object? Considered solely from the perspective of trade-secret law, the two objects were distinct, though from the perspective of copyright there was perhaps a case for infringement, although AT&T did not rely on copyright as much as on trade secret. From a technical perspective, the functions and processes that the software accomplishes are the same, but the means by which they are coded to do so are different. And from a pedagogical standpoint, the two are identical—they exemplify certain core features of an operating system (file-system structure, memory paging, process management)—all the rest is optimization, or bells and whistles. Understanding the nature of forking requires also that UNIX be understood from a social perspective, that is, from the perspective of an operating system created and modified by user-developers around the world according to particular and partial demands. It forms the basis for the emergence of a robust recursive public.

One of the more important instances of the forking of UNIX's perambulatory source code and the developing community of UNIX co-developers is the story of the Berkeley Software Distribution and its incorporation of the TCP/IP protocols. In 1975 Ken Thompson took a sabbatical in his hometown of Berkeley, California, where he helped members of the computer-science department with their installations of UNIX, arriving with V6 and the "50 bug fixes" diff tape. Ken had begun work on a compiler for the Pascal programming language that would run on UNIX, and this work was taken up by two young graduate students: Bill Joy and Chuck Hartley. (Joy would later co-found Sun Microsystems, one of the most successful UNIX-based workstation companies in the history of the industry.)

Joy, above nearly all others, enthusiastically participated in the informal distribution of source code. With a popular and well-built Pascal system, and a new text editor called ex (later vi), he created the Berkeley Software Distribution (BSD), a set of tools that could be used in combination with the UNIX operating system. They were extensions to the original UNIX operating system, but not a complete, rewritten version that might replace it. By all accounts, Joy served as a kind of one-man software-distribution house, making tapes and posting them, taking orders and cashing checks—all in

addition to creating software.[33] UNIX users around the world soon learned of this valuable set of extensions to the system, and before long, many were differentiating between AT&T UNIX and BSD UNIX.

According to Don Libes, Bell Labs allowed Berkeley to distribute its extensions to UNIX so long as the recipients also had a license from Bell Labs for the original UNIX (an arrangement similar to the one that governed Lions's *Commentary*).[34] From about 1976 until about 1981, BSD slowly became an independent distribution—indeed, a complete version of UNIX—well-known for the vi editor and the Pascal compiler, but also for the addition of virtual memory and its implementation on DEC's VAX machines.[35] It should be clear that the unusual quasi-commercial status of AT&T's UNIX allowed for this situation in a way that a fully commercial computer corporation would never have allowed. Consider, for instance, the fact that many UNIX users—students at a university, for instance—could not essentially know whether they were using an AT&T product or something called BSD UNIX created at Berkeley. The operating system functioned in the same way and, except for the presence of copyright notices that occasionally flashed on the screen, did not make any show of asserting its brand identity (that would come later, in the 1980s). Whereas a commercial computer manufacturer would have allowed something like BSD only if it were incorporated into and distributed as a single, marketable, and identifiable product with a clever name, AT&T turned something of a blind eye to the proliferation and spread of AT&T UNIX and the result were forks in the project: distinct bodies of source code, each an instance of something called UNIX.

As BSD developed, it gained different kinds of functionality than the UNIX from which it was spawned. The most significant development was the inclusion of code that allowed it to connect computers to the Arpanet, using the TCP/IP protocols designed by Vinton Cerf and Robert Kahn. The TCP/IP protocols were a key feature of the Arpanet, overseen by the Information Processing and Techniques Office (IPTO) of the Defense Advanced Research Projects Agency (DARPA) from its inception in 1967 until about 1977. The goal of the protocols was to allow different networks, each with its own machines and administrative boundaries, to be connected to each other.[36] Although there is a common heritage—in the form of J. C. R. Licklider—which ties the imagination of the time-sharing operat-

ing system to the creation of the "galactic network," the Arpanet initially developed completely independent of UNIX.[37] As a time-sharing operating system, UNIX was meant to allow the sharing of resources on a *single* computer, whether mainframe or minicomputer, but it was not initially intended to be connected to a network of other computers running UNIX, as is the case today.[38] The goal of Arpanet, by contrast, was explicitly to achieve the sharing of resources located on diverse machines across diverse networks.

To achieve the benefits of TCP/IP, the resources needed to be implemented in all of the different operating systems that were connected to the Arpanet—whatever operating system and machine happened to be in use at each of the nodes. However, by 1977, the original machines used on the network were outdated and increasingly difficult to maintain and, according to Kirk McKusick, the greatest expense was that of porting the old protocol software to new machines. Hence, IPTO decided to pursue in part a strategy of achieving coordination at the operating-system level, and they chose UNIX as one of the core platforms on which to standardize. In short, they had seen the light of portability. In about 1978 IPTO granted a contract to Bolt, Beranek, and Newman (BBN), one of the original Arpanet contractors, to integrate the TCP/IP protocols into the UNIX operating system.

But then something odd happened, according to Salus: "An initial prototype was done by BBN and given to Berkeley. Bill [Joy] immediately started hacking on it because it would only run an Ethernet at about 56K/sec utilizing 100% of the CPU on a 750. . . . Bill lobotomized the code and increased its performance to on the order of 700KB/sec. This caused some consternation with BBN when they came in with their 'finished' version, and Bill wouldn't accept it. There were battles for years after, about which version would be in the system. The Berkeley version ultimately won."[39]

Although it is not clear, it appears BBN intended to give Joy the code in order to include it in his BSD version of UNIX for distribution, and that Joy and collaborators intended to cooperate with Rob Gurwitz of BBN on a final implementation, but Berkeley insisted on "improving" the code to make it perform more to their needs, and BBN apparently dissented from this.[40] One result of this scuffle between BSD and BBN was a genuine fork: two bodies of code that did the same thing, competing with each other to become the standard UNIX implementation of TCP/IP. Here, then, was a

case of sharing source code that led to the creation of different versions of software—sharing without collaboration. Some sites used the BBN code, some used the Berkeley code.

Forking, however, does not imply permanent divergence, and the continual improvement, porting, and sharing of software can have odd consequences when forks occur. On the one hand, there are *particular pieces of source code*: they must be identifiable and exact, and prepended with a copyright notice, as was the case of the Berkeley code, which was famously and vigorously policed by the University of California regents, who allowed for a very liberal distribution of BSD code on the condition that the copyright notice was retained. On the other hand, there are particular named *collections of code* that work together (e.g., UNIX™, or DARPA-approved UNIX, or later, Certified Open Source [sm]) and are often identified by a trademark symbol intended, legally speaking, to differentiate products, not to assert ownership of particular instances of a product.

The odd consequence is this: Bill Joy's specific TCP/IP code was incorporated not only into BSD UNIX, but also into other versions of UNIX, including the UNIX distributed by AT&T (which had originally licensed UNIX to Berkeley) with the Berkeley copyright notice removed. This bizarre, tangled bank of licenses and code resulted in a famous suit and countersuit between AT&T and Berkeley, in which the intricacies of this situation were sorted out.[41] An innocent bystander, expecting UNIX to be a single thing, might be surprised to find that it takes different forms for reasons that are all but impossible to identify, but the cause of which is clear: different versions of sharing in conflict with one another; different moral and technical imaginations of order that result in complex entanglements of value and code.

The BSD fork of UNIX (and the subfork of TCP/IP) was only one of many to come. By the early 1980s, a proliferation of UNIX forks had emerged and would be followed shortly by a very robust commercialization. At the same time, the circulation of source code started to slow, as corporations began to compete by adding features and creating hardware specifically designed to run UNIX (such as the Sun Sparc workstation and the Solaris operating system, the result of Joy's commercialization of BSD in the 1980s). The question of how to make all of these versions work together eventually became the subject of the open-systems discussions that would dominate the workstation and networking sectors of the computer

market from the early 1980s to 1993, when the dual success of Windows NT and the arrival of the Internet into public consciousness changed the fortunes of the UNIX industry.

A second, and more important, effect of the struggle between BBN and BSD was simply the widespread adoption of the TCP/IP protocols. An estimated 98 percent of computer-science departments in the United States and many such departments around the world incorporated the TCP/IP protocols into their UNIX systems and gained instant access to Arpanet.[42] The fact that this occurred when it did is important: a few years later, during the era of the commercialization of UNIX, these protocols might very well not have been widely implemented (or more likely implemented in incompatible, nonstandard forms) by manufacturers, whereas before 1983, university computer scientists saw every benefit in doing so if it meant they could easily connect to the largest single computer network on the planet. The large, already functioning, relatively standard implementation of TCP/IP on UNIX (and the ability to look at the source code) gave these protocols a tremendous advantage in terms of their survival and success as the basis of a global and singular network.

Conclusion

The UNIX operating system is not just a technical achievement; it is the creation of a set of norms for sharing source code in an unusual environment: quasi-commercial, quasi-academic, networked, and planetwide. Sharing UNIX source code has taken three basic forms: porting source code (transferring it from one machine to another); teaching source code, or "porting" it to students in a pedagogical setting where the use of an actual working operating system vastly facilitates the teaching of theory and concepts; and forking source code (modifying the existing source code to do something new or different). This play of proliferation and differentiation is essential to the remarkably stable identity of UNIX, but that identity exists in multiple forms: technical (as a functioning, self-compatible operating system), legal (as a license-circumscribed version subject to intellectual property and commercial law), and pedagogical (as a conceptual exemplar, the paradigm of an operating system). Source code shared in this manner is essentially unlike any other kind of

source code in the world of computers, whether academic or commercial. It raises troubling questions about standardization, about control and audit, and about legitimacy that haunts not only UNIX but the Internet and its various "open" protocols as well.

Sharing source code in Free Software looks the way it does today because of UNIX. But UNIX looks the way it does not because of the inventive genius of Thompson and Ritchie, or the marketing and management brilliance of AT&T, but because *sharing produces its own kind of order*: operating systems and social systems. The fact that geeks are wont to speak of "the UNIX philosophy" means that UNIX is not just an operating system but a way of organizing the complex relations of life and work through technical means; a way of charting and breaching the boundaries between the academic, the aesthetic, and the commercial; a way of implementing ideas of a moral and technical order. What's more, as source code comes to include more and more of the activities of everyday communication and creation—as it comes to replace writing and supplement thinking—the genealogy of its portability and the history of its forking will illuminate the kinds of order emerging in practices and technologies far removed from operating systems—but tied intimately to the UNIX philosophy.

- # Intro - introduction to user commands

Type	Document
Date Added	10/24/2016, 1:18:00 PM
Modified	10/24/2016, 1:18:30 PM

Notes:

- # INTRO(1) Linux User's Manual

  Online documentation

  ```
  man intro
  ```

  Or PDF from man page

  ```
  man -troff -troff-device=ps intro | ps2pdf - > intro-manpage.pdf
  ```

Attachments

- intro-manpage.pdf

NAME

intro − introduction to user commands

DESCRIPTION

Section 1 of the manual describes user commands and tools, for example, file manipulation tools, shells, compilers, web browsers, file and image viewers and editors, and so on.

All commands yield a status value on termination. This value can be tested (e.g., in most shells the variable *$?* contains the status of the last executed command) to see whether the command completed successfully. A zero exit status is conventionally used to indicate success, and a nonzero status means that the command was unsuccessful. (Details of the exit status can be found in **wait**(2).) A nonzero exit status can be in the range 1 to 255, and some commands use different nonzero status values to indicate the reason why the command failed.

NOTES

Linux is a flavor of UNIX, and as a first approximation all user commands under UNIX work precisely the same under Linux (and FreeBSD and lots of other UNIX-like systems).

Under Linux, there are GUIs (graphical user interfaces), where you can point and click and drag, and hopefully get work done without first reading lots of documentation. The traditional UNIX environment is a CLI (command line interface), where you type commands to tell the computer what to do. That is faster and more powerful, but requires finding out what the commands are. Below a bare minimum, to get started.

Login

In order to start working, you probably first have to login, that is, give your username and password. See also **login**(1). The program *login* now starts a *shell* (command interpreter) for you. In case of a graphical login, you get a screen with menus or icons and a mouse click will start a shell in a window. See also **xterm**(1).

The shell

One types commands to the *shell*, the command interpreter. It is not built-in, but is just a program and you can change your shell. Everybody has her own favorite one. The standard one is called *sh*. See also **ash**(1), **bash**(1), **csh**(1), **zsh**(1), **chsh**(1).

A session might go like

knuth login: *aeb*
Password: ********
% *date*
Tue Aug 6 23:50:44 CEST 2002
% *cal*
August 2002
Su Mo Tu We Th Fr Sa
1 2 3
4 5 6 7 8 9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30 31

% *ls*
bin tel
% *ls −l*
total 2
drwxrwxr−x 2 aeb 1024 Aug 6 23:51 bin
−rw−rw−r−− 1 aeb 37 Aug 6 23:52 tel
% *cat tel*
maja 0501−1136285

```
peter   0136−7399214
% cp tel tel2
% ls −l
total 3
drwxr−xr−x   2 aeb      1024 Aug  6 23:51 bin
−rw−r−−r−−   1 aeb        37 Aug  6 23:52 tel
−rw−r−−r−−   1 aeb        37 Aug  6 23:53 tel2
% mv tel tel1
% ls −l
total 3
drwxr−xr−x   2 aeb      1024 Aug  6 23:51 bin
−rw−r−−r−−   1 aeb        37 Aug  6 23:52 tel1
−rw−r−−r−−   1 aeb        37 Aug  6 23:53 tel2
% diff tel1 tel2
% rm tel1
% grep maja tel2
maja   0501−1136285
%
```

and here typing Control-D ended the session. The **%** here was the command prompt—it is the shell's way of indicating that it is ready for the next command. The prompt can be customized in lots of ways, and one might include stuff like username, machine name, current directory, time, and so on. An assignment PS1="What next, master? " would change the prompt as indicated.

We see that there are commands *date* (that gives date and time), and *cal* (that gives a calendar).

The command *ls* lists the contents of the current directory—it tells you what files you have. With a *−l* option it gives a long listing, that includes the owner and size and date of the file, and the permissions people have for reading and/or changing the file. For example, the file "tel" here is 37 bytes long, owned by aeb and the owner can read and write it, others can only read it. Owner and permissions can be changed by the commands *chown* and *chmod*.

The command *cat* will show the contents of a file. (The name is from "concatenate and print": all files given as parameters are concatenated and sent to "standard output", here the terminal screen.)

The command *cp* (from "copy") will copy a file. On the other hand, the command *mv* (from "move") only renames it.

The command *diff* lists the differences between two files. Here there was no output because there were no differences.

The command *rm* (from "remove") deletes the file, and be careful! it is gone. No wastepaper basket or anything. Deleted means lost.

The command *grep* (from "g/re/p") finds occurrences of a string in one or more files. Here it finds Maja's telephone number.

Pathnames and the current directory

Files live in a large tree, the file hierarchy. Each has a *pathname* describing the path from the root of the tree (which is called /) to the file. For example, such a full pathname might be /home/aeb/tel. Always using full pathnames would be inconvenient, and the name of a file in the current directory may be abbreviated by giving only the last component. That is why "/home/aeb/tel" can be abbreviated to "tel" when the current directory is "/home/aeb".

The command *pwd* prints the current directory.

The command *cd* changes the current directory. Try "cd /" and "pwd" and "cd" and "pwd".

Directories

The command *mkdir* makes a new directory.

The command *rmdir* removes a directory if it is empty, and complains otherwise.

The command *find* (with a rather baroque syntax) will find files with given name or other properties. For example, "find . −name tel" would find the file "tel" starting in the present directory (which is called "."). And "find / −name tel" would do the same, but starting at the root of the tree. Large searches on a multi-GB disk will be time-consuming, and it may be better to use **locate**(1).

Disks and filesystems

The command *mount* will attach the filesystem found on some disk (or floppy, or CDROM or so) to the big filesystem hierarchy. And *umount* detaches it again. The command *df* will tell you how much of your disk is still free.

Processes

On a UNIX system many user and system processes run simultaneously. The one you are talking to runs in the *foreground*, the others in the *background*. The command *ps* will show you which processes are active and what numbers these processes have. The command *kill* allows you to get rid of them. Without option this is a friendly request: please go away. And "kill −9" followed by the number of the process is an immediate kill. Foreground processes can often be killed by typing Control-C.

Getting information

There are thousands of commands, each with many options. Traditionally commands are documented on *man pages*, (like this one), so that the command "man kill" will document the use of the command "kill" (and "man man" document the command "man"). The program *man* sends the text through some *pager*, usually *less*. Hit the space bar to get the next page, hit q to quit.

In documentation it is customary to refer to man pages by giving the name and section number, as in **man**(1). Man pages are terse, and allow you to find quickly some forgotten detail. For newcomers an introductory text with more examples and explanations is useful.

A lot of GNU/FSF software is provided with info files. Type "info info" for an introduction on the use of the program "info".

Special topics are often treated in HOWTOs. Look in */usr/share/doc/howto/en* and use a browser if you find HTML files there.

SEE ALSO

standards(7)

COLOPHON

This page is part of release 3.74 of the Linux *man-pages* project. A description of the project, information about reporting bugs, and the latest version of this page, can be found at http://www.kernel.org/doc/man−pages/.

- # The Linux Command Line

Type	Book
Author	William Shotts
URL	http://linuxcommand.org/
Date Added	10/24/2016, 7:55:48 PM
Modified	10/24/2016, 7:57:03 PM

Notes:

- # Command Line Self help

  Shotts, William. *The Linux Command Line*, n.d. http://linuxcommand.org/.

  ```
  pdftk The\ Linux\ Command\ Line\ -\ William\ Shotts.pdf cat 26-77 output The\ Linux\ Command\ Line-extract.pdf
  ```

Attachments

- The Linux Command Line - EXTRACT.pdf

1 – What Is The Shell?

When we speak of the command line, we are really referring to the *shell*. The shell is a program that takes keyboard commands and passes them to the operating system to carry out. Almost all Linux distributions supply a shell program from the GNU Project called `bash`. The name “bash” is an acronym for “Bourne Again SHell”, a reference to the fact `bash` is an enhanced replacement for `sh`, the original Unix shell program written by Steve Bourne.

Terminal Emulators

When using a graphical user interface, we need another program called a *terminal emulator* to interact with the shell. If we look through our desktop menus, we will probably find one. KDE uses `konsole` and GNOME uses `gnome-terminal`, though it's likely called simply “terminal” on our menu. There are a number of other terminal emulators available for Linux, but they all basically do the same thing; give us access to the shell. You will probably develop a preference for one or another based on the number of bells and whistles it has.

Your First Keystrokes

So let's get started. Launch the terminal emulator! Once it comes up, we should see something like this:

```
[me@linuxbox ~]$
```

This is called a *shell prompt* and it will appear whenever the shell is ready to accept input. While it may vary in appearance somewhat depending on the distribution, it will usually include your *username@machinename*, followed by the current working directory (more about that in a little bit) and a dollar sign.

If the last character of the prompt is a pound sign (“#”) rather than a dollar sign, the terminal session has *superuser* privileges. This means either we are logged in as the root user or we selected a terminal emulator that provides superuser (administrative) privi-

leges.

Assuming that things are good so far, let's try some typing. Enter some gibberish at the prompt like so:

```
[me@linuxbox ~]$ kaekfjaeifj
```

Since this command makes no sense, the shell will tell us so and give us another chance:

```
bash: kaekfjaeifj: command not found
[me@linuxbox ~]$
```

Command History

If we press the up-arrow key, we will see that the previous command “kaekfjaeifj” reappears after the prompt. This is called *command history*. Most Linux distributions remember the last 500 commands by default. Press the down-arrow key and the previous command disappears.

Cursor Movement

Recall the previous command with the up-arrow key again. Now try the left and right-arrow keys. See how we can position the cursor anywhere on the command line? This makes editing commands easy.

A Few Words About Mice And Focus

While the shell is all about the keyboard, you can also use a mouse with your terminal emulator. There is a mechanism built into the X Window System (the underlying engine that makes the GUI go) that supports a quick copy and paste technique. If you highlight some text by holding down the left mouse button and dragging the mouse over it (or double clicking on a word), it is copied into a buffer maintained by X. Pressing the middle mouse button will cause the text to be pasted at the cursor location. Try it.
Note: Don't be tempted to use `Ctrl-c` and `Ctrl-v` to perform copy and paste inside a terminal window. They don't work. These control codes have different meanings to the shell and were assigned many years before Microsoft Windows.

Your graphical desktop environment (most likely KDE or GNOME), in an effort to behave like Windows, probably has its *focus policy* set to "click to focus." This means for a window to get focus (become active) you need to click on it. This is contrary to the traditional X behavior of "focus follows mouse" which means that a window gets focus just by passing the mouse over it. The window will not come to the foreground until you click on it but it will be able to receive input. Setting the focus policy to "focus follows mouse" will make the copy and paste technique even more useful. Give it a try if you can (some desktop environments such as Ubuntu's Unity no longer support it). I think if you give it a chance you will prefer it. You will find this setting in the configuration program for your window manager.

Try Some Simple Commands

Now that we have learned to type, let's try a few simple commands. The first one is `date`. This command displays the current time and date.

```
[me@linuxbox ~]$ date
Thu Oct 25 13:51:54 EDT 2007
```

A related command is `cal` which, by default, displays a calendar of the current month.

```
[me@linuxbox ~]$ cal
    October 2007
Su Mo Tu We Th Fr Sa
    1  2  3  4  5  6
 7  8  9 10 11 12 13
14 15 16 17 18 19 20
21 22 23 24 25 26 27
28 29 30 31
```

To see the current amount of free space on your disk drives, enter `df`:

```
[me@linuxbox ~]$ df
Filesystem           1K-blocks      Used Available Use% Mounted on
/dev/sda2             15115452   5012392   9949716  34% /
/dev/sda5             59631908  26545424  30008432  47% /home
/dev/sda1               147764     17370    122765  13% /boot
```

```
tmpfs                    256856         0    256856   0% /dev/shm
```

Likewise, to display the amount of free memory, enter the `free` command.

```
[me@linuxbox ~]$ free
             total       used       free     shared    buffers     cached
Mem:    513712     503976       9736          0       5312     122916
-/+ buffers/cache: 375748     137964
Swap:  1052248     104712     947536
```

Ending A Terminal Session

We can end a terminal session by either closing the terminal emulator window, or by entering the `exit` command at the shell prompt:

```
[me@linuxbox ~]$ exit
```

The Console Behind The Curtain

Even if we have no terminal emulator running, several terminal sessions continue to run behind the graphical desktop. Called *virtual terminals* or *virtual consoles*, these sessions can be accessed on most Linux distributions by pressing `Ctrl-Alt-F1` through `Ctrl-Alt-F6`. When a session is accessed, it presents a login prompt into which we can enter our username and password. To switch from one virtual console to another, press `Alt` and `F1-F6`. To return to the graphical desktop, press `Alt-F7`.

Summing Up

As we begin our journey, we are introduced to the shell and see the command line for the first time and learn how to start and end a terminal session. We also see how to issue some simple commands and perform a little light command line editing. That wasn't so scary was it?

Further Reading

- To learn more about Steve Bourne, father of the Bourne Shell, see this Wikipedia article:
  http://en.wikipedia.org/wiki/Steve_Bourne
- Here is an article about the concept of shells in computing:
  http://en.wikipedia.org/wiki/Shell_(computing)

2 – Navigation

The first thing we need to learn (besides just typing) is how to navigate the file system on our Linux system. In this chapter we will introduce the following commands:

- `pwd` - Print name of current working directory
- `cd` - Change directory
- `ls` - List directory contents

Understanding The File System Tree

Like Windows, a Unix-like operating system such as Linux organizes its files in what is called a *hierarchical directory structure*. This means that they are organized in a tree-like pattern of directories (sometimes called folders in other systems), which may contain files and other directories. The first directory in the file system is called the *root directory*. The root directory contains files and subdirectories, which contain more files and subdirectories and so on and so on.

Note that unlike Windows, which has a separate file system tree for each storage device, Unix-like systems such as Linux always have a single file system tree, regardless of how many drives or storage devices are attached to the computer. Storage devices are attached (or more correctly, *mounted*) at various points on the tree according to the whims of the *system administrator*, the person (or persons) responsible for the maintenance of the system.

The Current Working Directory

Most of us are probably familiar with a graphical file manager which represents the file system tree as in Figure 1. Notice that the tree is usually shown upended, that is, with the root at the top and the various branches descending below.

However, the command line has no pictures, so to navigate the file system tree we need to think of it in a different way.

Imagine that the file system is a maze shaped like an upside-down tree and we are able to

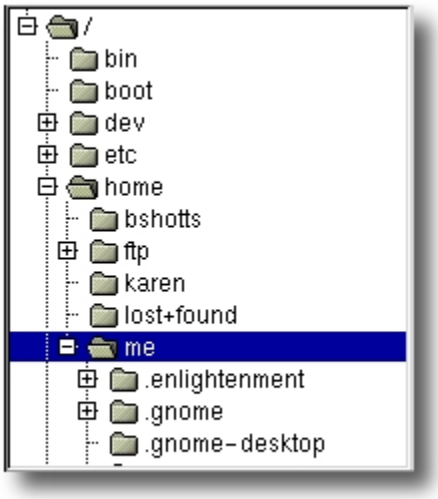


Figure 1: File system tree as shown by a graphical file manager

stand in the middle of it. At any given time, we are inside a single directory and we can see the files contained in the directory and the pathway to the directory above us (called the *parent directory*) and any subdirectories below us. The directory we are standing in is called the *current working directory*. To display the current working directory, we use the `pwd` (print working directory) command.

```
[me@linuxbox ~]$ pwd
/home/me
```

When we first log in to our system (or start a terminal emulator session) our current working directory is set to our *home directory*. Each user account is given its own home directory and it is the only place a regular user is allowed to write files.

Listing The Contents Of A Directory

To list the files and directories in the current working directory, we use the `ls` command.

```
[me@linuxbox ~]$ ls
Desktop  Documents  Music  Pictures  Public  Templates  Videos
```

Actually, we can use the `ls` command to list the contents of any directory, not just the current working directory, and there are many other fun things it can do as well. We'll spend more time with `ls` in the next chapter.

Changing The Current Working Directory

To change your working directory (where we are standing in our tree-shaped maze) we use the `cd` command. To do this, type `cd` followed by the *pathname* of the desired working directory. A pathname is the route we take along the branches of the tree to get to the directory we want. Pathnames can be specified in one of two different ways; as *absolute pathnames* or as *relative pathnames*. Let's deal with absolute pathnames first.

Absolute Pathnames

An absolute pathname begins with the root directory and follows the tree branch by branch until the path to the desired directory or file is completed. For example, there is a directory on your system in which most of your system's programs are installed. The pathname of the directory is `/usr/bin`. This means from the root directory (represented by the leading slash in the pathname) there is a directory called "usr" which contains a directory called "bin".

```
[me@linuxbox ~]$ cd /usr/bin
[me@linuxbox bin]$ pwd
/usr/bin
[me@linuxbox bin]$ ls


...Listing of many, many files ...
```

Now we can see that we have changed the current working directory to `/usr/bin` and that it is full of files. Notice how the shell prompt has changed? As a convenience, it is usually set up to automatically display the name of the working directory.

Relative Pathnames

Where an absolute pathname starts from the root directory and leads to its destination, a relative pathname starts from the working directory. To do this, it uses a couple of special symbols to represent relative positions in the file system tree. These special symbols are "." (dot) and ".." (dot dot).

The "." symbol refers to the working directory and the ".." symbol refers to the working directory's parent directory. Here is how it works. Let's change the working directory to

`/usr/bin` again:

```
[me@linuxbox ~]$ cd /usr/bin
[me@linuxbox bin]$ pwd
/usr/bin
```

Okay, now let's say that we wanted to change the working directory to the parent of `/usr/bin` which is `/usr`. We could do that two different ways. Either with an absolute pathname:

```
[me@linuxbox bin]$ cd /usr
[me@linuxbox usr]$ pwd
/usr
```

Or, with a relative pathname:

```
[me@linuxbox bin]$ cd ..
[me@linuxbox usr]$ pwd
/usr
```

Two different methods with identical results. Which one should we use? The one that requires the least typing!

Likewise, we can change the working directory from `/usr` to `/usr/bin` in two different ways. Either using an absolute pathname:

```
[me@linuxbox usr]$ cd /usr/bin
[me@linuxbox bin]$ pwd
/usr/bin
```

Or, with a relative pathname:

```
[me@linuxbox usr]$ cd ./bin
[me@linuxbox bin]$ pwd
/usr/bin
```

Now, there is something important that I must point out here. In almost all cases, you can

omit the "./". It is implied. Typing:

```
[me@linuxbox usr]$ cd bin
```

does the same thing. In general, if you do not specify a pathname to something, the working directory will be assumed.

Some Helpful Shortcuts

In Table 2-1 we see some useful ways the current working directory can be quickly changed.

Table 2-1: cd Shortcuts

Shortcut	Result
`cd`	Changes the working directory to your home directory.
`cd -`	Changes the working directory to the previous working directory.
`cd ~`*`user_name`*	Changes the working directory to the home directory of *user_name*. For example, `cd ~bob` will change the directory to the home directory of user “bob.”

Important Facts About Filenames

1. Filenames that begin with a period character are hidden. This only means that `ls` will not list them unless you say `ls -a`. When your account was created, several hidden files were placed in your home directory to configure things for your account. Later on we will take a closer look at some of these files to see how you can customize your environment. In addition, some applications place their configuration and settings files in your home directory as hidden files.
2. Filenames and commands in Linux, like Unix, are case sensitive. The filenames “File1” and “file1” refer to different files.
3. Linux has no concept of a “file extension” like some other operating systems. You may name files any way you like. The contents and/or purpose of a file is determined by other means. Although Unix-like operating system don’t use

file extensions to determine the contents/purpose of files, some application programs do.

4. Though Linux supports long filenames which may contain embedded spaces and punctuation characters, limit the punctuation characters in the names of files you create to period, dash, and underscore. *Most importantly, do not embed spaces in filenames.* If you want to represent spaces between words in a filename, use underscore characters. You will thank yourself later.

Summing Up

In this chapter we saw how the shell treats the directory structure of the system. We learned about absolute and relative pathnames and the basic commands that are used to move about that structure. In the next chapter we will use this knowledge to go on a tour of a modern Linux system.

3 – Exploring The System

Now that we know how to move around the file system, it's time for a guided tour of our Linux system. Before we start however, we're going to learn some more commands that will be useful along the way:

- `ls` – List directory contents
- `file` – Determine file type
- `less` – View file contents

More Fun With `ls`

The `ls` command is probably the most used command, and for good reason. With it, we can see directory contents and determine a variety of important file and directory attributes. As we have seen, we can simply enter `ls` to see a list of files and subdirectories contained in the current working directory:

```
[me@linuxbox ~]$ ls
Desktop  Documents  Music  Pictures  Public  Templates  Videos
```

Besides the current working directory, we can specify the directory to list, like so:

```
me@linuxbox ~]$ ls /usr
bin  games    kerberos  libexec  sbin   src
etc  include  lib       local    share  tmp
```

Or even specify multiple directories. In this example we will list both the user's home directory (symbolized by the "~" character) and the `/usr` directory:

```
[me@linuxbox ~]$ ls ~ /usr
/home/me:
```

```
Desktop  Documents  Music  Pictures  Public  Templates  Videos


/usr:
bin  games    kerberos  libexec  sbin   src
etc  include  lib       local    share  tmp
```

We can also change the format of the output to reveal more detail:

```
[me@linuxbox ~]$ ls -l
total 56
drwxrwxr-x 2 me me 4096 2007-10-26 17:20 Desktop
drwxrwxr-x 2 me me 4096 2007-10-26 17:20 Documents
drwxrwxr-x 2 me me 4096 2007-10-26 17:20 Music
drwxrwxr-x 2 me me 4096 2007-10-26 17:20 Pictures
drwxrwxr-x 2 me me 4096 2007-10-26 17:20 Public
drwxrwxr-x 2 me me 4096 2007-10-26 17:20 Templates
drwxrwxr-x 2 me me 4096 2007-10-26 17:20 Videos
```

By adding “-l” to the command, we changed the output to the long format.

Options And Arguments

This brings us to a very important point about how most commands work. Commands are often followed by one or more *options* that modify their behavior, and further, by one or more *arguments*, the items upon which the command acts. So most commands look kind of like this:

```
command -options arguments
```

Most commands use options consisting of a single character preceded by a dash, for example, “-l”, but many commands, including those from the GNU Project, also support *long options*, consisting of a word preceded by two dashes. Also, many commands allow multiple short options to be strung together. In this example, the `ls` command is given two options, the “l” option to produce long format output, and the “t” option to sort the result by the file's modification time.

```
[me@linuxbox ~]$ ls -lt
```

We'll add the long option “--reverse” to reverse the order of the sort:

```
[me@linuxbox ~]$ ls -lt --reverse
```

Note that command options, like filenames in Linux, are case-sensitive.

The `ls` command has a large number of possible options. The most common are listed in Table 3-1.

Table 3- 1: Common ls Options

Option	Long Option	Description
`-a`	`--all`	List all files, even those with names that begin with a period, which are normally not listed (i.e., hidden).
`-A`	`--almost-all`	Like the `-a` option above except it does not list `.` (current directory) and `..` (parent directory).
`-d`	`--directory`	Ordinarily, if a directory is specified, `ls` will list the contents of the directory, not the directory itself. Use this option in conjunction with the `-l` option to see details about the directory rather than its contents.
`-F`	`--classify`	This option will append an indicator character to the end of each listed name. For example, a “/” if the name is a directory.
`-h`	`--human-readable`	In long format listings, display file sizes in human readable format rather than in bytes.
`-l`		Display results in long format.
`-r`	`--reverse`	Display the results in reverse order. Normally, `ls` displays its results in ascending alphabetical order.
`-S`		Sort results by file size.
`-t`		Sort by modification time.

A Longer Look At Long Format

As we saw before, the “-l” option causes `ls` to display its results in long format. This format contains a great deal of useful information. Here is the `Examples` directory from an Ubuntu system:

```
-rw-r--r-- 1 root root 3576296 2007-04-03 11:05 Experience ubuntu.ogg
-rw-r--r-- 1 root root 1186219 2007-04-03 11:05 kubuntu-leaflet.png
-rw-r--r-- 1 root root   47584 2007-04-03 11:05 logo-Edubuntu.png
-rw-r--r-- 1 root root   44355 2007-04-03 11:05 logo-Kubuntu.png
-rw-r--r-- 1 root root   34391 2007-04-03 11:05 logo-Ubuntu.png
-rw-r--r-- 1 root root   32059 2007-04-03 11:05 oo-cd-cover.odf
-rw-r--r-- 1 root root  159744 2007-04-03 11:05 oo-derivatives.doc
-rw-r--r-- 1 root root   27837 2007-04-03 11:05 oo-maxwell.odt
-rw-r--r-- 1 root root   98816 2007-04-03 11:05 oo-trig.xls
-rw-r--r-- 1 root root  453764 2007-04-03 11:05 oo-welcome.odt
-rw-r--r-- 1 root root  358374 2007-04-03 11:05 ubuntu Sax.ogg
```

Let's look at the different fields from one of the files and examine their meanings:

Table 3-2: ls Long Listing Fields

Field	Meaning
`-rw-r--r--`	Access rights to the file. The first character indicates the type of file. Among the different types, a leading dash means a regular file, while a “d” indicates a directory. The next three characters are the access rights for the file's owner, the next three are for members of the file's group, and the final three are for everyone else. The full meaning of this is discussed in Chapter 9 – Permissions.
`1`	File's number of hard links. See the discussion of links later in this chapter.
`root`	The username of the file's owner.
`root`	The name of the group which owns the file.
`32059`	Size of the file in bytes.
`2007-04-03 11:05`	Date and time of the file's last modification.
`oo-cd-cover.odf`	Name of the file.

Determining A File's Type With `file`

As we explore the system it will be useful to know what files contain. To do this we will use the `file` command to determine a file's type. As we discussed earlier, filenames in Linux are not required to reflect a file's contents. While a filename like “picture.jpg” would normally be expected to contain a JPEG compressed image, it is not required to in Linux. We can invoke the `file` command this way:

```
file filename
```

When invoked, the `file` command will print a brief description of the file's contents. For example:

```
[me@linuxbox ~]$ file picture.jpg
picture.jpg: JPEG image data, JFIF standard 1.01
```

There are many kinds of files. In fact, one of the common ideas in Unix-like operating systems such as Linux is that “everything is a file.” As we proceed with our lessons, we will see just how true that statement is.

While many of the files on your system are familiar, for example MP3 and JPEG, there are many kinds that are a little less obvious and a few that are quite strange.

Viewing File Contents With `less`

The `less` command is a program to view text files. Throughout our Linux system, there are many files that contain human-readable text. The `less` program provides a convenient way to examine them.

What Is “Text”?

There are many ways to represent information on a computer. All methods involve defining a relationship between the information and some numbers that will be used to represent it. Computers, after all, only understand numbers and all data is converted to numeric representation.

Some of these representation systems are very complex (such as compressed video files), while others are rather simple. One of the earliest and simplest is called *ASCII text*. ASCII (pronounced "As-Key") is short for American Standard

Code for Information Interchange. This is a simple encoding scheme that was first used on Teletype machines to map keyboard characters to numbers.
Text is a simple one-to-one mapping of characters to numbers. It is very compact. Fifty characters of text translates to fifty bytes of data. It is important to understand that text only contains a simple mapping of characters to numbers. It is not the same as a word processor document such as one created by Microsoft Word or OpenOffice.org Writer. Those files, in contrast to simple ASCII text, contain many non-text elements that are used to describe its structure and formatting. Plain ASCII text files contain only the characters themselves and a few rudimentary control codes like tabs, carriage returns and line feeds.
Throughout a Linux system, many files are stored in text format and there are many Linux tools that work with text files. Even Windows recognizes the importance of this format. The well-known NOTEPAD.EXE program is an editor for plain ASCII text files.

Why would we want to examine text files? Because many of the files that contain system settings (called *configuration files*) are stored in this format, and being able to read them gives us insight about how the system works. In addition, many of the actual programs that the system uses (called *scripts*) are stored in this format. In later chapters, we will learn how to edit text files in order to modify systems settings and write our own scripts, but for now we will just look at their contents.

The `less` command is used like this:

```
less filename
```

Once started, the `less` program allows you to scroll forward and backward through a text file. For example, to examine the file that defines all the system's user accounts, enter the following command:

```
[me@linuxbox ~]$ less /etc/passwd
```

Once the `less` program starts, we can view the contents of the file. If the file is longer than one page, we can scroll up and down. To exit `less`, press the “q” key.

The table below lists the most common keyboard commands used by `less`.

Table 3-3: less Commands

Command	Action
Page Up or `b`	Scroll back one page
Page Down or space	Scroll forward one page
Up Arrow	Scroll up one line
Down Arrow	Scroll down one line
`G`	Move to the end of the text file
`1G` or `g`	Move to the beginning of the text file
`/`*characters*	Search forward to the next occurrence of *characters*
`n`	Search for the next occurrence of the previous search
`h`	Display help screen
`q`	Quit `less`

Less Is More

The `less` program was designed as an improved replacement of an earlier Unix program called `more`. The name "less" is a play on the phrase "less is more"—a motto of modernist architects and designers.
`less` falls into the class of programs called "pagers," programs that allow the easy viewing of long text documents in a page by page manner. Whereas the `more` program could only page forward, the `less` program allows paging both forward and backward and has many other features as well.

A Guided Tour

The file system layout on your Linux system is much like that found on other Unix-like systems. The design is actually specified in a published standard called the *Linux Filesystem Hierarchy Standard*. Not all Linux distributions conform to the standard exactly but most come pretty close.

Next, we are going to wander around the file system ourselves to see what makes our Linux system tick. This will give you a chance to practice your navigation skills. One of the things we will discover is that many of the interesting files are in plain human-readable text. As we go about our tour, try the following:

1. `cd` into a given directory
2. List the directory contents with `ls -l`
3. If you see an interesting file, determine its contents with `file`
4. If it looks like it might be text, try viewing it with `less`

Remember the copy and paste trick! If you are using a mouse, you can double click on a filename to copy it and middle click to paste it into commands.

As we wander around, don't be afraid to look at stuff. Regular users are largely prohibited from messing things up. That's the system administrators job! If a command complains about something, just move on to something else. Spend some time looking around. The system is ours to explore. Remember, in Linux, there are no secrets!

Table 3-4 lists just a few of the directories we can explore. Feel free to try more!

Table 3-4: Directories Found On Linux Systems

Directory	Comments
`/`	The root directory. Where everything begins.
`/bin`	Contains binaries (programs) that must be present for the system to boot and run.
`/boot`	Contains the Linux kernel, initial RAM disk image (for drivers needed at boot time), and the boot loader. Interesting files: • `/boot/grub/grub.conf` or `menu.lst`, which are used to configure the boot loader. • `/boot/vmlinuz`, the Linux kernel
`/dev`	This is a special directory which contains *device nodes*. "Everything is a file" also applies to devices. Here is where the kernel maintains a list of all the devices it understands.

Directory	Comments
`/etc`	The `/etc` directory contains all of the system-wide configuration files. It also contains a collection of shell scripts which start each of the system services at boot time. Everything in this directory should be readable text. Interesting files: While everything in `/etc` is interesting, here are some of my all-time favorites: • `/etc/crontab`, a file that defines when automated jobs will run. • `/etc/fstab`, a table of storage devices and their associated mount points. • `/etc/passwd`, a list of the user accounts.
`/home`	In normal configurations, each user is given a directory in `/home`. Ordinary users can only write files in their home directories. This limitation protects the system from errant user activity.
`/lib`	Contains shared library files used by the core system programs. These are similar to DLLs in Windows.
`/lost+found`	Each formatted partition or device using a Linux file system, such as ext3, will have this directory. It is used in the case of a partial recovery from a file system corruption event. Unless something really bad has happened to your system, this directory will remain empty.
`/media`	On modern Linux systems the `/media` directory will contain the mount points for removable media such as USB drives, CD-ROMs, etc. that are mounted automatically at insertion.
`/mnt`	On older Linux systems, the `/mnt` directory contains mount points for removable devices that have been mounted manually.
`/opt`	The `/opt` directory is used to install “optional” software. This is mainly used to hold commercial software products that may be installed on your system.

Directory	Comments
/proc	The /proc directory is special. It's not a real file system in the sense of files stored on your hard drive. Rather, it is a virtual file system maintained by the Linux kernel. The "files" it contains are peepholes into the kernel itself. The files are readable and will give you a picture of how the kernel sees your computer.
/root	This is the home directory for the root account.
/sbin	This directory contains "system" binaries. These are programs that perform vital system tasks that are generally reserved for the superuser.
/tmp	The /tmp directory is intended for storage of temporary, transient files created by various programs. Some configurations cause this directory to be emptied each time the system is rebooted.
/usr	The /usr directory tree is likely the largest one on a Linux system. It contains all the programs and support files used by regular users.
/usr/bin	/usr/bin contains the executable programs installed by your Linux distribution. It is not uncommon for this directory to hold thousands of programs.
/usr/lib	The shared libraries for the programs in /usr/bin.
/usr/local	The /usr/local tree is where programs that are not included with your distribution but are intended for system-wide use are installed. Programs compiled from source code are normally installed in /usr/local/bin. On a newly installed Linux system, this tree exists, but it will be empty until the system administrator puts something in it.
/usr/sbin	Contains more system administration programs.
/usr/share	/usr/share contains all the shared data used by programs in /usr/bin. This includes things like default configuration files, icons, screen backgrounds, sound files, etc.
/usr/share/doc	Most packages installed on the system will include some kind of documentation. In /usr/share/doc, we will find documentation files organized by package.

Directory	Comments
/var	With the exception of /tmp and /home, the directories we have looked at so far remain relatively static, that is, their contents don't change. The /var directory tree is where data that is likely to change is stored. Various databases, spool files, user mail, etc. are located here.
/var/log	/var/log contains *log files*, records of various system activity. These are very important and should be monitored from time to time. The most useful one is /var/log/messages. Note that for security reasons on some systems, you must be the superuser to view log files .

Symbolic Links

As we look around, we are likely to see a directory listing with an entry like this:

```
lrwxrwxrwx 1 root root   11 2007-08-11 07:34 libc.so.6 -> libc-2.6.so
```

Notice how the first letter of the listing is “l” and the entry seems to have two filenames? This is a special kind of a file called a *symbolic link* (also known as a *soft link* or *sym-link*.) In most Unix-like systems it is possible to have a file referenced by multiple names. While the value of this may not be obvious, it is really a useful feature.

Picture this scenario: A program requires the use of a shared resource of some kind contained in a file named “foo,” but “foo” has frequent version changes. It would be good to include the version number in the filename so the administrator or other interested party could see what version of “foo” is installed. This presents a problem. If we change the name of the shared resource, we have to track down every program that might use it and change it to look for a new resource name every time a new version of the resource is installed. That doesn't sound like fun at all.

Here is where symbolic links save the day. Let's say we install version 2.6 of “foo,” which has the filename “foo-2.6” and then create a symbolic link simply called “foo” that points to “foo-2.6.” This means that when a program opens the file “foo”, it is actually opening the file “foo-2.6”. Now everybody is happy. The programs that rely on “foo” can find it and we can still see what actual version is installed. When it is time to upgrade to “foo-2.7,” we just add the file to our system, delete the symbolic link “foo” and create a new one that points to the new version. Not only does this solve the problem of the version upgrade, but it also allows us to keep both versions on our machine. Imagine that “foo-2.7” has a bug (damn those developers!) and we need to revert to the old version. Again, we just delete the symbolic link pointing to the new version and create a new

symbolic link pointing to the old version.

The directory listing above (from the `/lib` directory of a Fedora system) shows a symbolic link called “libc.so.6” that points to a shared library file called “libc-2.6.so.” This means that programs looking for “libc.so.6” will actually get the file “libc-2.6.so.” We will learn how to create symbolic links in the next chapter.

Hard Links

While we are on the subject of links, we need to mention that there is a second type of link called a *hard link*. Hard links also allow files to have multiple names, but they do it in a different way. We’ll talk more about the differences between symbolic and hard links in the next chapter.

Summing Up

With our tour behind us, we have learned a lot about our system. We've seen various files and directories and their contents. One thing you should take away from this is how open the system is. In Linux there are many important files that are plain human-readable text. Unlike many proprietary systems, Linux makes everything available for examination and study.

Further Reading

- The full version of the *Linux Filesystem Hierarchy Standard* can be found here: http://www.pathname.com/fhs/
- An article about the directory structure of Unix and Unix-like systems: http://en.wikipedia.org/wiki/Unix_directory_structure
- A detailed description of the ASCII text format: http://en.wikipedia.org/wiki/ASCII

4 – Manipulating Files And Directories

At this point, we are ready for some real work! This chapter will introduce the following commands:

- `cp` – Copy files and directories
- `mv` – Move/rename files and directories
- `mkdir` – Create directories
- `rm` – Remove files and directories
- `ln` – Create hard and symbolic links

These five commands are among the most frequently used Linux commands. They are used for manipulating both files and directories.

Now, to be frank, some of the tasks performed by these commands are more easily done with a graphical file manager. With a file manager, we can drag and drop a file from one directory to another, cut and paste files, delete files, etc. So why use these old command line programs?

The answer is power and flexibility. While it is easy to perform simple file manipulations with a graphical file manager, complicated tasks can be easier with the command line programs. For example, how could we copy all the HTML files from one directory to another, but only copy files that do not exist in the destination directory or are newer than the versions in the destination directory? Pretty hard with a file manager. Pretty easy with the command line:

```
cp -u *.html destination
```

Wildcards

Before we begin using our commands, we need to talk about a shell feature that makes these commands so powerful. Since the shell uses filenames so much, it provides special characters to help you rapidly specify groups of filenames. These special characters are

called *wildcards*. Using wildcards (which is also known as *globbing*) allow you to select filenames based on patterns of characters. The table below lists the wildcards and what they select:

Table 4-1: Wildcards

Wildcard	Meaning
`*`	Matches any characters
`?`	Matches any single character
`[`*`characters`*`]`	Matches any character that is a member of the set *characters*
`[!`*`characters`*`]`	Matches any character that is not a member of the set *characters*
`[[:`*`class`*`:]]`	Matches any character that is a member of the specified *class*

Table 4-2 lists the most commonly used character classes:

Table 4-2: Commonly Used Character Classes

Character Class	Meaning
`[:alnum:]`	Matches any alphanumeric character
`[:alpha:]`	Matches any alphabetic character
`[:digit:]`	Matches any numeral
`[:lower:]`	Matches any lowercase letter
`[:upper:]`	Matches any uppercase letter

Using wildcards makes it possible to construct very sophisticated selection criteria for filenames. Here are some examples of patterns and what they match:

Table 4-3: Wildcard Examples

Pattern	Matches
`*`	All files
`g*`	Any file beginning with “g”
`b*.txt`	Any file beginning with “b” followed by any characters and ending with “.txt”

`Data???`	Any file beginning with “Data” followed by exactly three characters
`[abc]*`	Any file beginning with either an “a”, a “b”, or a “c”
`BACKUP.[0-9][0-9][0-9]`	Any file beginning with “BACKUP.” followed by exactly three numerals
`[[:upper:]]*`	Any file beginning with an uppercase letter
`[![:digit:]]*`	Any file not beginning with a numeral
`*[[:lower:]123]`	Any file ending with a lowercase letter or the numerals “1”, “2”, or “3”

Wildcards can be used with any command that accepts filenames as arguments, but we’ll talk more about that in Chapter 7.

Character Ranges

If you are coming from another Unix-like environment or have been reading some other books on this subject, you may have encountered the `[A-Z]` or the `[a-z]` character range notations. These are traditional Unix notations and worked in older versions of Linux as well. They can still work, but you have to be very careful with them because they will not produce the expected results unless properly configured. For now, you should avoid using them and use character classes instead.

Wildcards Work In The GUI Too

Wildcards are especially valuable not only because they are used so frequently on the command line, but are also supported by some graphical file managers.

- In **Nautilus** (the file manager for GNOME), you can select files using the Edit/Select Pattern menu item. Just enter a file selection pattern with wildcards and the files in the currently viewed directory will be highlighted for selection.
- In some versions of **Dolphin** and **Konqueror** (the file managers for KDE), you can enter wildcards directly on the location bar. For example, if you want to see all the files starting with a lowercase “u” in the /usr/bin directory, enter “/usr/bin/u*” in the location bar and it will display the result.

> Many ideas originally found in the command line interface make their way into the graphical interface, too. It is one of the many things that make the Linux desktop so powerful.

`mkdir` – Create Directories

The `mkdir` command is used to create directories. It works like this:

```
mkdir directory...
```

A note on notation: When three periods follow an argument in the description of a command (as above), it means that the argument can be repeated, thus:

```
mkdir dir1
```

would create a single directory named “dir1”, while

```
mkdir dir1 dir2 dir3
```

would create three directories named “dir1”, “dir2”, and “dir3”.

`cp` – Copy Files And Directories

The `cp` command copies files or directories. It can be used two different ways:

```
cp item1 item2
```

to copy the single file or directory “item1” to file or directory “item2” and:

```
cp item... directory
```

to copy multiple items (either files or directories) into a directory.

Useful Options And Examples

Here are some of the commonly used options (the short option and the equivalent long option) for `cp`:

Table 4-4: cp Options

Option	Meaning
`-a, --archive`	Copy the files and directories and all of their attributes, including ownerships and permissions. Normally, copies take on the default attributes of the user performing the copy.
`-i, --interactive`	Before overwriting an existing file, prompt the user for confirmation. **If this option is not specified, `cp` will silently overwrite files.**
`-r, --recursive`	Recursively copy directories and their contents. This option (or the `-a` option) is required when copying directories.
`-u, --update`	When copying files from one directory to another, only copy files that either don't exist, or are newer than the existing corresponding files, in the destination directory.
`-v, --verbose`	Display informative messages as the copy is performed.

Table 4-5: cp Examples

Command	Results
`cp` *`file1 file2`*	Copy *file1* to *file2*. **If *file2* exists, it is overwritten with the contents of *file1*.** If *file2* does not exist, it is created.
`cp -i` *`file1 file2`*	Same as above, except that if *file2* exists, the user is prompted before it is overwritten.
`cp` *`file1 file2 dir1`*	Copy *file1* and *file2* into directory *dir1*. *dir1* must already exist.
`cp dir1/* dir2`	Using a wildcard, all the files in *dir1* are copied into *dir2*. *dir2* must already exist.

`cp -r dir1 dir2`	Copy the contents of directory *dir1* to directory *dir2*. If directory *dir2* does not exist, it is created and, after the copy, will contain the same contents as directory *dir1*. If directory *dir2* does exist, then directory *dir1* (and its contents) will be copied into *dir2*.

`mv` – Move And Rename Files

The `mv` command performs both file moving and file renaming, depending on how it is used. In either case, the original filename no longer exists after the operation. `mv` is used in much the same way as `cp`:

```
mv item1 item2
```

to move or rename file or directory “item1” to “item2” or:

```
mv item... directory
```

to move one or more items from one directory to another.

Useful Options And Examples

`mv` shares many of the same options as `cp`:

Table 4-6: mv Options

Option	Meaning
`-i, --interactive`	Before overwriting an existing file, prompt the user for confirmation. **If this option is not specified, `mv` will silently overwrite files.**
`-u, --update`	When moving files from one directory to another, only move files that either don't exist, or are newer than the existing corresponding files in the destination directory.
`-v, --verbose`	Display informative messages as the move is

	performed.

Table 4-7: mv Examples

Command	Results
`mv file1 file2`	Move *file1* to *file2*. **If *file2* exists, it is overwritten with the contents of *file1*.** If *file2* does not exist, it is created. **In either case, *file1* ceases to exist.**
`mv -i file1 file2`	Same as above, except that if *file2* exists, the user is prompted before it is overwritten.
`mv file1 file2 dir1`	Move *file1* and *file2* into directory *dir1*. *dir1* must already exist.
`mv dir1 dir2`	If directory *dir2* does not exist, create directory *dir2* and move the contents of directory *dir1* into *dir2* and delete directory *dir1*. If directory *dir2* does exist, move directory *dir1* (and its contents) into directory *dir2*.

`rm` – Remove Files And Directories

The `rm` command is used to remove (delete) files and directories:

```
rm item...
```

where “item” is one or more files or directories.

Useful Options And Examples

Here are some of the common options for `rm`:

Table 4-8: rm Options

Option	Meaning
`-i, --interactive`	Before deleting an existing file, prompt the user for confirmation. **If this option is not specified, `rm` will silently delete files.**

`-r, --recursive`	Recursively delete directories. This means that if a directory being deleted has subdirectories, delete them too. To delete a directory, this option must be specified.
`-f, --force`	Ignore nonexistent files and do not prompt. This overrides the `--interactive` option.
`-v, --verbose`	Display informative messages as the deletion is performed.

Table 4-9: rm Examples

Command	Results
`rm` *`file1`*	Delete *file1* silently.
`rm -i` *`file1`*	Same as above, except that the user is prompted for confirmation before the deletion is performed.
`rm -r` *`file1 dir1`*	Delete *file1* and *dir1* and its contents.
`rm -rf` *`file1 dir1`*	Same as above, except that if either *file1* or *dir1* do not exist, `rm` will continue silently.

Be Careful With `rm`!

Unix-like operating systems such as Linux do not have an undelete command. Once you delete something with `rm`, it's gone. Linux assumes you're smart and you know what you're doing.

Be particularly careful with wildcards. Consider this classic example. Let's say you want to delete just the HTML files in a directory. To do this, you type:

```
rm *.html
```

which is correct, but if you accidentally place a space between the "*" and the ".html" like so:

```
rm * .html
```

the `rm` command will delete all the files in the directory and then complain that there is no file called ".html".

Here is a useful tip. Whenever you use wildcards with `rm` (besides carefully checking your typing!), test the wildcard first with `ls`. This will let you see the

files that will be deleted. Then press the up arrow key to recall the command and replace the `ls` with `rm`.

`ln` – Create Links

The `ln` command is used to create either hard or symbolic links. It is used in one of two ways:

```
ln file link
```

to create a hard link, and:

```
ln -s item link
```

to create a symbolic link where “item” is either a file or a directory.

Hard Links

Hard links are the original Unix way of creating links, compared to symbolic links, which are more modern. By default, every file has a single hard link that gives the file its name. When we create a hard link, we create an additional directory entry for a file. Hard links have two important limitations:

1. A hard link cannot reference a file outside its own file system. This means a link cannot reference a file that is not on the same disk partition as the link itself.
2. A hard link may not reference a directory.

A hard link is indistinguishable from the file itself. Unlike a symbolic link, when you list a directory containing a hard link you will see no special indication of the link. When a hard link is deleted, the link is removed but the contents of the file itself continue to exist (that is, its space is not deallocated) until all links to the file are deleted.

It is important to be aware of hard links because you might encounter them from time to time, but modern practice prefers symbolic links, which we will cover next.

Symbolic Links

Symbolic links were created to overcome the limitations of hard links. Symbolic links work by creating a special type of file that contains a text pointer to the referenced file or

directory. In this regard, they operate in much the same way as a Windows shortcut though of course, they predate the Windows feature by many years ;-)

A file pointed to by a symbolic link, and the symbolic link itself are largely indistinguishable from one another. For example, if you write something to the symbolic link, the referenced file is written to. However when you delete a symbolic link, only the link is deleted, not the file itself. If the file is deleted before the symbolic link, the link will continue to exist, but will point to nothing. In this case, the link is said to be *broken*. In many implementations, the `ls` command will display broken links in a distinguishing color, such as red, to reveal their presence.

The concept of links can seem very confusing, but hang in there. We're going to try all this stuff and it will, hopefully, become clear.

Let's Build A Playground

Since we are going to do some real file manipulation, let's build a safe place to “play” with our file manipulation commands. First we need a directory to work in. We'll create one in our home directory and call it “playground.”

Creating Directories

The `mkdir` command is used to create a directory. To create our playground directory we will first make sure we are in our home directory and will then create the new directory:

```
[me@linuxbox ~]$ cd
[me@linuxbox ~]$ mkdir playground
```

To make our playground a little more interesting, let's create a couple of directories inside it called “dir1” and “dir2”. To do this, we will change our current working directory to `playground` and execute another `mkdir`:

```
[me@linuxbox ~]$ cd playground
[me@linuxbox playground]$ mkdir dir1 dir2
```

Notice that the `mkdir` command will accept multiple arguments allowing us to create both directories with a single command.

Copying Files

Next, let's get some data into our playground. We'll do this by copying a file. Using the

cp command, we'll copy the passwd file from the /etc directory to the current working directory:

```
[me@linuxbox playground]$ cp /etc/passwd .
```

Notice how we used the shorthand for the current working directory, the single trailing period. So now if we perform an ls, we will see our file:

```
[me@linuxbox playground]$ ls -l
total 12
drwxrwxr-x 2 me  me 4096 2008-01-10 16:40 dir1
drwxrwxr-x 2 me  me 4096 2008-01-10 16:40 dir2
-rw-r--r-- 1 me  me 1650 2008-01-10 16:07 passwd
```

Now, just for fun, let's repeat the copy using the “-v” option (verbose) to see what it does:

```
[me@linuxbox playground]$ cp -v /etc/passwd .
`/etc/passwd' -> `./passwd'
```

The cp command performed the copy again, but this time displayed a concise message indicating what operation it was performing. Notice that cp overwrote the first copy without any warning. Again this is a case of cp assuming that you know what you’re are doing. To get a warning, we'll include the “-i” (interactive) option:

```
[me@linuxbox playground]$ cp -i /etc/passwd .
cp: overwrite `./passwd'?
```

Responding to the prompt by entering a “y” will cause the file to be overwritten, any other character (for example, “n”) will cause cp to leave the file alone.

Moving And Renaming Files

Now, the name “passwd” doesn't seem very playful and this is a playground, so let's change it to something else:

```
[me@linuxbox playground]$ mv passwd fun
```

Let's pass the fun around a little by moving our renamed file to each of the directories and back again:

```
[me@linuxbox playground]$ mv fun dir1
```

to move it first to directory `dir1`, then:

```
[me@linuxbox playground]$ mv dir1/fun dir2
```

to move it from `dir1` to `dir2`, then:

```
[me@linuxbox playground]$ mv dir2/fun .
```

to finally bring it back to the current working directory. Next, let's see the effect of `mv` on directories. First we will move our data file into `dir1` again:

```
[me@linuxbox playground]$ mv fun dir1
```

then move `dir1` into `dir2` and confirm it with `ls`:

```
[me@linuxbox playground]$ mv dir1 dir2
[me@linuxbox playground]$ ls -l dir2
total 4
drwxrwxr-x 2 me  me   4096 2008-01-11 06:06 dir1
[me@linuxbox playground]$ ls -l dir2/dir1
total 4
-rw-r--r-- 1 me  me   1650 2008-01-10 16:33 fun
```

Note that since `dir2` already existed, `mv` moved `dir1` into `dir2`. If `dir2` had not existed, `mv` would have renamed `dir1` to `dir2`. Lastly, let's put everything back:

```
[me@linuxbox playground]$ mv dir2/dir1 .
[me@linuxbox playground]$ mv dir1/fun .
```

Creating Hard Links

Now we'll try some links. First the hard links. We'll create some links to our data file like so:

```
[me@linuxbox playground]$ ln fun fun-hard
[me@linuxbox playground]$ ln fun dir1/fun-hard
[me@linuxbox playground]$ ln fun dir2/fun-hard
```

So now we have four instances of the file "fun". Let's take a look our playground directory:

```
[me@linuxbox playground]$ ls -l
total 16
drwxrwxr-x 2 me   me   4096 2008-01-14 16:17 dir1
drwxrwxr-x 2 me   me   4096 2008-01-14 16:17 dir2
-rw-r--r-- 4 me   me   1650 2008-01-10 16:33 fun
-rw-r--r-- 4 me   me   1650 2008-01-10 16:33 fun-hard
```

One thing you notice is that the second field in the listing for `fun` and `fun-hard` both contain a "4" which is the number of hard links that now exist for the file. You'll remember that a file will aways have at least one link because the file's name is created by a link. So, how do we know that `fun` and `fun-hard` are, in fact, the same file? In this case, `ls` is not very helpful. While we can see that `fun` and `fun-hard` are both the same size (field 5), our listing provides no way to be sure. To solve this problem, we're going to have to dig a little deeper.

When thinking about hard links, it is helpful to imagine that files are made up of two parts: the data part containing the file's contents and the name part which holds the file's name. When we create hard links, we are actually creating additional name parts that all refer to the same data part. The system assigns a chain of disk blocks to what is called an *inode*, which is then associated with the name part. Each hard link therefore refers to a specific inode containing the file's contents.

The `ls` command has a way to reveal this information. It is invoked with the "-i" option:

```
[me@linuxbox playground]$ ls -li
total 16
12353539 drwxrwxr-x 2 me   me   4096 2008-01-14 16:17 dir1
12353540 drwxrwxr-x 2 me   me   4096 2008-01-14 16:17 dir2
12353538 -rw-r--r-- 4 me   me   1650 2008-01-10 16:33 fun
```

```
12353538 -rw-r--r-- 4 me   me   1650 2008-01-10 16:33 fun-hard
```

In this version of the listing, the first field is the inode number and, as we can see, both `fun` and `fun-hard` share the same inode number, which confirms they are the same file.

Creating Symbolic Links

Symbolic links were created to overcome the two disadvantages of hard links: Hard links cannot span physical devices and hard links cannot reference directories, only files. Symbolic links are a special type of file that contains a text pointer to the target file or directory.

Creating symbolic links is similar to creating hard links:

```
[me@linuxbox playground]$ ln -s fun fun-sym
[me@linuxbox playground]$ ln -s ../fun dir1/fun-sym
[me@linuxbox playground]$ ln -s ../fun dir2/fun-sym
```

The first example is pretty straightforward, we simply add the “-s” option to create a symbolic link rather than a hard link. But what about the next two? Remember, when we create a symbolic link, we are creating a text description of where the target file is relative to the symbolic link. It's easier to see if we look at the `ls` output:

```
[me@linuxbox playground]$ ls -l dir1
total 4
-rw-r--r-- 4 me   me   1650 2008-01-10 16:33 fun-hard
lrwxrwxrwx 1 me   me      6 2008-01-15 15:17 fun-sym -> ../fun
```

The listing for `fun-sym` in `dir1` shows that it is a symbolic link by the leading “l” in the first field and that it points to “../fun”, which is correct. Relative to the location of `fun-sym`, `fun` is in the directory above it. Notice too, that the length of the symbolic link file is 6, the number of characters in the string “../fun” rather than the length of the file to which it is pointing.

When creating symbolic links, you can either use absolute pathnames:

```
ln -s /home/me/playground/fun dir1/fun-sym
```

or relative pathnames, as we did in our earlier example. Using relative pathnames is more desirable because it allows a directory containing symbolic links to be renamed and/or moved without breaking the links.

In addition to regular files, symbolic links can also reference directories:

```
[me@linuxbox playground]$ ln -s dir1 dir1-sym
[me@linuxbox playground]$ ls -l
total 16
drwxrwxr-x 2 me   me   4096 2008-01-15 15:17 dir1
lrwxrwxrwx 1 me   me      4 2008-01-16 14:45 dir1-sym -> dir1
drwxrwxr-x 2 me   me   4096 2008-01-15 15:17 dir2
-rw-r--r-- 4 me   me   1650 2008-01-10 16:33 fun
-rw-r--r-- 4 me   me   1650 2008-01-10 16:33 fun-hard
lrwxrwxrwx 1 me   me      3 2008-01-15 15:15 fun-sym -> fun
```

Removing Files And Directories

As we covered earlier, the `rm` command is used to delete files and directories. We are going to use it to clean up our playground a little bit. First, let's delete one of our hard links:

```
[me@linuxbox playground]$ rm fun-hard
[me@linuxbox playground]$ ls -l
total 12
drwxrwxr-x 2 me   me   4096 2008-01-15 15:17 dir1
lrwxrwxrwx 1 me   me      4 2008-01-16 14:45 dir1-sym -> dir1
drwxrwxr-x 2 me   me   4096 2008-01-15 15:17 dir2
-rw-r--r-- 3 me   me   1650 2008-01-10 16:33 fun
lrwxrwxrwx 1 me   me      3 2008-01-15 15:15 fun-sym -> fun
```

That worked as expected. The file `fun-hard` is gone and the link count shown for `fun` is reduced from four to three, as indicated in the second field of the directory listing. Next, we'll delete the file `fun`, and just for enjoyment, we'll include the “-i” option to show what that does:

```
[me@linuxbox playground]$ rm -i fun
rm: remove regular file `fun'?
```

Enter “y” at the prompt and the file is deleted. But let's look at the output of `ls` now. Noticed what happened to `fun-sym`? Since it's a symbolic link pointing to a now-nonexistent file, the link is *broken*:

```
[me@linuxbox playground]$ ls -l
total 8
drwxrwxr-x 2 me   me   4096 2008-01-15 15:17 dir1
lrwxrwxrwx 1 me   me      4 2008-01-16 14:45 dir1-sym -> dir1
drwxrwxr-x 2 me   me   4096 2008-01-15 15:17 dir2
lrwxrwxrwx 1 me   me      3 2008-01-15 15:15 fun-sym -> fun
```

Most Linux distributions configure `ls` to display broken links. On a Fedora box, broken links are displayed in blinking red text! The presence of a broken link is not, in and of itself dangerous but it is rather messy. If we try to use a broken link we will see this:

```
[me@linuxbox playground]$ less fun-sym
fun-sym: No such file or directory
```

Let's clean up a little. We'll delete the symbolic links:

```
[me@linuxbox playground]$ rm fun-sym dir1-sym
[me@linuxbox playground]$ ls -l
total 8
drwxrwxr-x 2 me   me   4096 2008-01-15 15:17 dir1
drwxrwxr-x 2 me   me   4096 2008-01-15 15:17 dir2
```

One thing to remember about symbolic links is that most file operations are carried out on the link's target, not the link itself. `rm` is an exception. When you delete a link, it is the link that is deleted, not the target.

Finally, we will remove our playground. To do this, we will return to our home directory and use rm with the recursive option (-r) to delete playground and all of its contents, including its subdirectories:

```
[me@linuxbox playground]$ cd
[me@linuxbox ~]$ rm -r playground
```

Creating Symlinks With The GUI

The file managers in both GNOME and KDE provide an easy and automatic method of creating symbolic links. With GNOME, holding the Ctrl+Shift keys

> while dragging a file will create a link rather than copying (or moving) the file. In KDE, a small menu appears whenever a file is dropped, offering a choice of copying, moving, or linking the file.

Summing Up

We've covered a lot of ground here and it will take a while to fully sink in. Perform the playground exercise over and over until it makes sense. It is important to get a good understanding of basic file manipulation commands and wildcards. Feel free to expand on the playground exercise by adding more files and directories, using wildcards to specify files for various operations. The concept of links is a little confusing at first, but take the time to learn how they work. They can be a real lifesaver.

Further Reading

- A discussion of symbolic links: http://en.wikipedia.org/wiki/Symbolic_link

5 – Working With Commands

Up to this point, we have seen a series of mysterious commands, each with its own mysterious options and arguments. In this chapter, we will attempt to remove some of that mystery and even create some of our own commands. The commands introduced in this chapter are:

- `type` – Indicate how a command name is interpreted
- `which` – Display which executable program will be executed
- `help` – Get help for shell builtins
- `man` – Display a command's manual page
- `apropos` – Display a list of appropriate commands
- `info` – Display a command's info entry
- `whatis` – Display a very brief description of a command
- `alias` – Create an alias for a command

What Exactly Are Commands?

A command can be one of four different things:

1. **An executable program** like all those files we saw in `/usr/bin`. Within this category, programs can be *compiled binaries* such as programs written in C and C++, or programs written in *scripting languages* such as the shell, perl, python, ruby, etc.
2. **A command built into the shell itself**. `bash` supports a number of commands internally called *shell builtins*. The `cd` command, for example, is a shell builtin.
3. **A shell function.** These are miniature shell scripts incorporated into the *environment*. We will cover configuring the environment and writing shell functions in later chapters, but for now, just be aware that they exist.
4. **An alias.** Commands that we can define ourselves, built from other commands.

Identifying Commands

It is often useful to know exactly which of the four kinds of commands is being used and Linux provides a couple of ways to find out.

`type` – Display A Command's Type

The `type` command is a shell builtin that displays the kind of command the shell will execute, given a particular command name. It works like this:

```
type command
```

where “command” is the name of the command you want to examine. Here are some examples:

```
[me@linuxbox ~]$ type type
type is a shell builtin
[me@linuxbox ~]$ type ls
ls is aliased to `ls --color=tty'
[me@linuxbox ~]$ type cp
cp is /bin/cp
```

Here we see the results for three different commands. Notice that the one for `ls` (taken from a Fedora system) and how the `ls` command is actually an alias for the `ls` command with the “-- color=tty” option added. Now we know why the output from `ls` is displayed in color!

`which` – Display An Executable's Location

Sometimes there is more than one version of an executable program installed on a system. While this is not very common on desktop systems, it's not unusual on large servers. To determine the exact location of a given executable, the `which` command is used:

```
[me@linuxbox ~]$ which ls
/bin/ls
```

`which` only works for executable programs, not builtins nor aliases that are substitutes for actual executable programs. When we try to use `which` on a shell builtin, for example, `cd`, we either get no response or an error message:

```
[me@linuxbox ~]$ which cd
/usr/bin/which: no cd in (/opt/jre1.6.0_03/bin:/usr/lib/qt-
3.3/bin:/usr/kerberos/bin:/opt/jre1.6.0_03/bin:/usr/lib/ccache:/usr/l
ocal/bin:/usr/bin:/bin:/home/me/bin)
```

which is a fancy way of saying “command not found.”

Getting A Command's Documentation

With this knowledge of what a command is, we can now search for the documentation available for each kind of command.

`help` – Get Help For Shell Builtins

`bash` has a built-in help facility available for each of the shell builtins. To use it, type “help” followed by the name of the shell builtin. For example:

```
[me@linuxbox ~]$ help cd
cd: cd [-L|[-P [-e]]] [dir]
Change the shell working directory.

Change the current directory to DIR.  The default DIR is the value of
the HOME shell variable.

The variable CDPATH defines the search path for the directory
containing DIR.  Alternative directory names in CDPATH are separated
by a colon (:). A null directory name is the same as the current
directory.  If DIR begins with a slash (/), then CDPATH is not used.

If the directory is not found, and the shell option `cdable_vars' is
set, the word is assumed to be  a variable name.  If that variable
has a value, its value is used for DIR.

Options:
-L   force symbolic links to be followed
-P   use the physical directory structure without following symbolic
     links
-e   if the -P option is supplied, and the current working directory
     cannot be determined successfully, exit with a non-zero status

The default is to follow symbolic links, as if `-L' were specified.

Exit Status:
Returns 0 if the directory is changed, and if $PWD is set
successfully when -P is used; non-zero otherwise.
```

A note on notation: When square brackets appear in the description of a command's syntax, they indicate optional items. A vertical bar character indicates mutually exclusive items. In the case of the `cd` command above:

```
cd [-L|[-P[-e]]] [dir]
```

This notation says that the command `cd` may be followed optionally by either a "-L" or a "-P" and further, if the "-P" option is specified the "-e" option may also be included followed by the optional argument "dir".

While the output of `help` for the `cd` commands is concise and accurate, it is by no means tutorial and as we can see, it also seems to mention a lot of things we haven't talked about yet! Don't worry. We'll get there.

`--help` – Display Usage Information

Many executable programs support a "--help" option that displays a description of the command's supported syntax and options. For example:

```
[me@linuxbox ~]$ mkdir --help
Usage: mkdir [OPTION] DIRECTORY...
Create the DIRECTORY(ies), if they do not already exist.

  -Z, --context=CONTEXT (SELinux) set security context to CONTEXT
Mandatory arguments to long options are mandatory for short options
too.
  -m, --mode=MODE   set file mode (as in chmod), not a=rwx – umask
  -p, --parents     no error if existing, make parent directories as
                    needed
  -v, --verbose     print a message for each created directory
      --help        display this help and exit
      --version     output version information and exit


Report bugs to <bug-coreutils@gnu.org>.
```

Some programs don't support the "--help" option, but try it anyway. Often it results in an error message that will reveal the same usage information.

`man` – Display A Program's Manual Page

Most executable programs intended for command line use provide a formal piece of documentation called a *manual* or *man page*. A special paging program called `man` is used to view them. It is used like this:

```
man program
```

where “program” is the name of the command to view.

Man pages vary somewhat in format but generally contain a title, a synopsis of the command's syntax, a description of the command's purpose, and a listing and description of each of the command's options. Man pages, however, do not usually include examples, and are intended as a reference, not a tutorial. As an example, let's try viewing the man page for the `ls` command:

```
[me@linuxbox ~]$ man ls
```

On most Linux systems, `man` uses `less` to display the manual page, so all of the familiar `less` commands work while displaying the page.

The “manual” that `man` displays is broken into sections and not only covers user commands but also system administration commands, programming interfaces, file formats and more. The table below describes the layout of the manual:

Table 5-1: Man Page Organization

Section	Contents
1	User commands
2	Programming interfaces kernel system calls
3	Programming interfaces to the C library
4	Special files such as device nodes and drivers
5	File formats
6	Games and amusements such as screen savers
7	Miscellaneous
8	System administration commands

Sometimes we need to look in a specific section of the manual to find what we are looking for. This is particularly true if we are looking for a file format that is also the name of a command. Without specifying a section number, we will always get the first instance of a match, probably in section 1. To specify a section number, we use `man` like this:

```
man section search_term
```

For example:

```
[me@linuxbox ~]$ man 5 passwd
```

This will display the man page describing the file format of the `/etc/passwd` file.

`apropos` – Display Appropriate Commands

It is also possible to search the list of man pages for possible matches based on a search term. It's very crude but sometimes helpful. Here is an example of a search for man pages using the search term “floppy”:

```
[me@linuxbox ~]$ apropos floppy
create_floppy_devices (8)  - udev callout to create all possible
                             floppy device based on the CMOS type
fdformat             (8)  - Low-level formats a floppy disk
floppy               (8)  - format floppy disks
gfloppy              (1)  - a simple floppy formatter for the GNOME
mbadblocks           (1)  - tests a floppy disk, and marks the bad
                            blocks in the FAT
mformat              (1)  - add an MSDOS filesystem to a low-level
                            formatted floppy disk
```

The first field in each line of output is the name of the man page, the second field shows the section. Note that the `man` command with the “-k” option performs the exact same function as `apropos`.

`whatis` – Display A Very Brief Description Of A Command

The `whatis` program displays the name and a one line description of a man page matching a specified keyword:

```
[me@linuxbox ~]$ whatis ls
ls                   (1)  - list directory contents
```

The Most Brutal Man Page Of Them All

As we have seen, the manual pages supplied with Linux and other Unix-like systems are intended as reference documentation and not as tutorials. Many man pages are hard to read, but I think that the grand prize for difficulty has got to go to the man page for `bash`. As I was doing my research for this book, I gave it careful review to ensure that I was covering most of its topics. When printed, it's over 80 pages long and extremely dense, and its structure makes absolutely no sense to a new user.

On the other hand, it is very accurate and concise, as well as being extremely complete. So check it out if you dare and look forward to the day when you can read it and it all makes sense.

`info` – Display A Program's Info Entry

The GNU Project provides an alternative to man pages for their programs, called “info.” Info pages are displayed with a reader program named, appropriately enough, `info`. Info pages are *hyperlinked* much like web pages. Here is a sample:

```
File: coreutils.info,  Node: ls invocation,  Next: dir invocation,
Up: Directory listing


10.1 `ls': List directory contents
==================================


The `ls' program lists information about files (of any type,
including directories).  Options and file arguments can be intermixed
arbitrarily, as usual.


   For non-option command-line arguments that are directories, by
default `ls' lists the contents of directories, not recursively, and
omitting files with names beginning with `.'.  For other non-option
arguments, by default `ls' lists just the filename.  If no non-option
argument is specified, `ls' operates on the current directory, acting
as if it had been invoked with a single argument of `.'.
```

```
   By default, the output is sorted alphabetically, according to the
--zz-Info: (coreutils.info.gz)ls invocation, 63 lines --Top---------
```

The `info` program reads *info files*, which are tree structured into individual *nodes*, each containing a single topic. Info files contain hyperlinks that can move you from node to node. A hyperlink can be identified by its leading asterisk, and is activated by placing the cursor upon it and pressing the enter key.

To invoke `info`, type “info” followed optionally by the name of a program. Below is a table of commands used to control the reader while displaying an info page:

Table 5-2: info Commands

Command	Action
?	Display command help
`PgUp` or `Backspace`	Display previous page
`PgDn` or `Space`	Display next page
n	Next - Display the next node
p	Previous - Display the previous node
u	Up - Display the parent node of the currently displayed node, usually a menu.
`Enter`	Follow the hyperlink at the cursor location
q	Quit

Most of the command line programs we have discussed so far are part of the GNU Project's “coreutils” package, so typing:

```
[me@linuxbox ~]$ info coreutils
```

will display a menu page with hyperlinks to each program contained in the coreutils package.

README And Other Program Documentation Files

Many software packages installed on your system have documentation files residing in the `/usr/share/doc` directory. Most of these are stored in plain text format and can

be viewed with `less`. Some of the files are in HTML format and can be viewed with a web browser. We may encounter some files ending with a ".gz" extension. This indicates that they have been compressed with the `gzip` compression program. The gzip package includes a special version of `less` called `zless` that will display the contents of gzip-compressed text files.

Creating Your Own Commands With `alias`

Now for our very first experience with programming! We will create a command of our own using the `alias` command. But before we start, we need to reveal a small command line trick. It's possible to put more than one command on a line by separating each command with a semicolon character. It works like this:

```
command1; command2; command3...
```

Here's the example we will use:

```
[me@linuxbox ~]$ cd /usr; ls; cd -
bin  games    kerberos  lib64    local  share  tmp
etc  include  lib       libexec  sbin   src
/home/me
[me@linuxbox ~]$
```

As we can see, we have combined three commands on one line. First we change directory to `/usr` then list the directory and finally return to the original directory (by using `'cd -'`) so we end up where we started. Now let's turn this sequence into a new command using `alias`. The first thing we have to do is dream up a name for our new command. Let's try "test". Before we do that, it would be a good idea to find out if the name "test" is already being used. To find out, we can use the `type` command again:

```
[me@linuxbox ~]$ type test
test is a shell builtin
```

Oops! The name "test" is already taken. Let's try "foo":

```
[me@linuxbox ~]$ type foo
bash: type: foo: not found
```

Great! “foo” is not taken. So let's create our alias:

```
[me@linuxbox ~]$ alias foo='cd /usr; ls; cd -'
```

Notice the structure of this command:

```
alias name='string'
```

After the command “alias” we give alias a name followed immediately (no whitespace allowed) by an equals sign, followed immediately by a quoted string containing the meaning to be assigned to the name. After we define our alias, it can be used anywhere the shell would expect a command. Let's try it:

```
[me@linuxbox ~]$ foo
bin  games    kerberos  lib64    local  share  tmp
etc  include  lib       libexec  sbin   src
/home/me
[me@linuxbox ~]$
```

We can also use the `type` command again to see our alias:

```
[me@linuxbox ~]$ type foo
foo is aliased to `cd /usr; ls ; cd -'
```

To remove an alias, the `unalias` command is used, like so:

```
[me@linuxbox ~]$ unalias foo
[me@linuxbox ~]$ type foo
bash: type: foo: not found
```

While we purposefully avoided naming our alias with an existing command name, it is not uncommon to do so. This is often done to apply a commonly desired option to each invocation of a common command. For instance, we saw earlier how the `ls` command is often aliased to add color support:

```
[me@linuxbox ~]$ type ls
ls is aliased to `ls --color=tty'
```

To see all the aliases defined in the environment, use the `alias` command without arguments. Here are some of the aliases defined by default on a Fedora system. Try and figure out what they all do:

```
[me@linuxbox ~]$ alias
alias l.='ls -d .* --color=tty'
alias ll='ls -l --color=tty'
alias ls='ls --color=tty'
```

There is one tiny problem with defining aliases on the command line. They vanish when your shell session ends. In a later chapter, we will see how to add our own aliases to the files that establish the environment each time we log on, but for now, enjoy the fact that we have taken our first, albeit tiny, step into the world of shell programming!

Summing Up

Now that we have learned how to find the documentation for commands, go and look up the documentation for all the commands we have encountered so far. Study what additional options are available and try them out!

Further Reading

There are many online sources of documentation for Linux and the command line. Here are some of the best:

- The *Bash Reference Manual* is a reference guide to the `bash` shell. It's still a reference work but contains examples and is easier to read than the `bash` man page.
  http://www.gnu.org/software/bash/manual/bashref.html
- The *Bash FAQ* contains answers to frequently asked questions regarding `bash`. This list is aimed at intermediate to advanced users, but contains a lot of good information.
  http://mywiki.wooledge.org/BashFAQ
- The GNU Project provides extensive documentation for its programs, which form the core of the Linux command line experience. You can see a complete list here:
  http://www.gnu.org/manual/manual.html
- Wikipedia has an interesting article on man pages:
  http://en.wikipedia.org/wiki/Man_page

6 – Redirection

In this lesson we are going to unleash what may be the coolest feature of the command line. It's called *I/O redirection*. The "I/O" stands for *input/output* and with this facility you can redirect the input and output of commands to and from files, as well as connect multiple commands together into powerful command *pipelines*. To show off this facility, we will introduce the following commands:

- `cat` - Concatenate files
- `sort` - Sort lines of text
- `uniq` - Report or omit repeated lines
- `grep` - Print lines matching a pattern
- `wc` - Print newline, word, and byte counts for each file
- `head` - Output the first part of a file
- `tail` - Output the last part of a file
- `tee` - Read from standard input and write to standard output and files

Standard Input, Output, And Error

Many of the programs that we have used so far produce output of some kind. This output often consists of two types. First, we have the program's results; that is, the data the program is designed to produce, and second, we have status and error messages that tell us how the program is getting along. If we look at a command like `ls`, we can see that it displays its results and its error messages on the screen.

Keeping with the Unix theme of "everything is a file," programs such as `ls` actually send their results to a special file called *standard output* (often expressed as *stdout*) and their status messages to another file called s*tandard error* (*stderr*). By default, both standard output and standard error are linked to the screen and not saved into a disk file.

In addition, many programs take input from a facility called *standard input* (*stdin*) which is, by default, attached to the keyboard.

- ## In the Beginning was the Command Line

Type	Document
Author	Neal Stephenson
URL	http://www.cryptonomicon.com/beginning.html
Date Added	10/24/2016, 7:33:26 PM
Modified	10/24/2016, 7:34:12 PM

Notes:

- # Rambling on the command line

  Stephenson, Neal. â€œIn the Beginning Was the Command Line,â€ n.d. http://www.cryptonomicon.com/beginning.html.

  Sections: Linux, Memento Mori

  ```
  pandoc command.txt -f markdown -t latex --standalone | pdflatex
  ```

Attachments

- command.pdf

In the Beginning was the Command Line

by Neal Stephenson

LINUX

During the late 1980's and early 1990's I spent a lot of time programming Macintoshes, and eventually decided for fork over several hundred dollars for an Apple product called the Macintosh Programmer's Workshop, or MPW. MPW had competitors, but it was unquestionably the premier software development system for the Mac. It was what Apple's own engineers used to write Macintosh code. Given that MacOS was far more technologically advanced, at the time, than its competition, and that Linux did not even exist yet, and given that this was the actual program used by Apple's world-class team of creative engineers, I had high expectations. It arrived on a stack of floppy disks about a foot high, and so there was plenty of time for my excitement to build during the endless installation process. The first time I launched MPW, I was probably expecting some kind of touch-feely multimedia showcase. Instead it was austere, almost to the point of being intimidating. It was a scrolling window into which you could type simple, unformatted text. The system would then interpret these lines of text as commands, and try to execute them.

It was, in other words, a glass teletype running a command line interface. It came with all sorts of cryptic but powerful commands, which could be invoked by typing their names, and which I learned to use only gradually. It was not until a few years later, when I began messing around with Unix, that I understood that the command line interface embodied in MPW was a re-creation of Unix.

In other words, the first thing that Apple's hackers had done when they'd got the MacOS up and running–probably even before they'd gotten it up and running–was to re-create the Unix interface, so that they would be able to get some useful work done. At the time, I simply couldn't get my mind around this, but: as far as Apple's hackers were concerned, the Mac's vaunted Graphical User Interface was an impediment, something to be circumvented before the little toaster even came out onto the market.

Even before my Powerbook crashed and obliterated my big file in July 1995, there had been danger signs. An old college buddy of mine, who starts and runs high-tech companies in Boston, had developed a commercial product using Macintoshes as the front end. Basically the Macs were high-performance graphics terminals, chosen for their sweet user interface, giving users access to a large database of graphical information stored on a network of much more powerful, but less user-friendly, computers. This fellow was the second person who turned me on to Macintoshes, by the way, and through the mid-1980's we had shared the thrill of being high-tech cognoscenti, using superior Apple technology in a world of DOS-using knuckleheads. Early versions of my friend's system had worked

well, he told me, but when several machines joined the network, mysterious crashes began to occur; sometimes the whole network would just freeze. It was one of those bugs that could not be reproduced easily. Finally they figured out that these network crashes were triggered whenever a user, scanning the menus for a particular item, held down the mouse button for more than a couple of seconds.

Fundamentally, the MacOS could only do one thing at a time. Drawing a menu on the screen is one thing. So when a menu was pulled down, the Macintosh was not capable of doing anything else until that indecisive user released the button.

This is not such a bad thing in a single-user, single-process machine (although it's a fairly bad thing), but it's no good in a machine that is on a network, because being on a network implies some kind of continual low-level interaction with other machines. By failing to respond to the network, the Mac caused a network-wide crash.

In order to work with other computers, and with networks, and with various different types of hardware, an OS must be incomparably more complicated and powerful than either MS-DOS or the original MacOS. The only way of connecting to the Internet that's worth taking seriously is PPP, the Point-to-Point Protocol, which (never mind the details) makes your computer–temporarily–a full-fledged member of the Global Internet, with its own unique address, and various privileges, powers, and responsibilities appertaining thereunto. Technically it means your machine is running the TCP/IP protocol, which, to make a long story short, revolves around sending packets of data back and forth, in no particular order, and at unpredictable times, according to a clever and elegant set of rules. But sending a packet of data is one thing, and so an OS that can only do one thing at a time cannot simultaneously be part of the Internet and do anything else. When TCP/IP was invented, running it was an honor reserved for Serious Computers–mainframes and high-powered minicomputers used in technical and commercial settings–and so the protocol is engineered around the assumption that every computer using it is a serious machine, capable of doing many things at once. Not to put too fine a point on it, a Unix machine. Neither MacOS nor MS-DOS was originally built with that in mind, and so when the Internet got hot, radical changes had to be made.

When my Powerbook broke my heart, and when Word stopped recognizing my old files, I jumped to Unix. The obvious alternative to MacOS would have been Windows. I didn't really have anything against Microsoft, or Windows. But it was pretty obvious, now, that old PC operating systems were overreaching, and showing the strain, and, perhaps, were best avoided until they had learned to walk and chew gum at the same time.

The changeover took place on a particular day in the summer of 1995. I had been San Francisco for a couple of weeks, using my PowerBook to work on a document. The document was too big to fit onto a single floppy, and so I hadn't made a backup since leaving home. The PowerBook crashed and wiped out the

entire file.

It happened just as I was on my way out the door to visit a company called Electric Communities, which in those days was in Los Altos. I took my PowerBook with me. My friends at Electric Communities were Mac users who had all sorts of utility software for unerasing files and recovering from disk crashes, and I was certain I could get most of the file back.

As it turned out, two different Mac crash recovery utilities were unable to find any trace that my file had ever existed. It was completely and systematically wiped out. We went through that hard disk block by block and found disjointed fragments of countless old, discarded, forgotten files, but none of what I wanted. The metaphor shear was especially brutal that day. It was sort of like watching the girl you've been in love with for ten years get killed in a car wreck, and then attending her autopsy, and learning that underneath the clothes and makeup she was just flesh and blood.

I must have been reeling around the offices of Electric Communities in some kind of primal Jungian fugue, because at this moment three weirdly synchronistic things happened.

(1) Randy Farmer, a co-founder of the company, came in for a quick visit along with his family–he was recovering from back surgery at the time. He had some hot gossip: "Windows 95 mastered today." What this meant was that Microsoft's new operating system had, on this day, been placed on a special compact disk known as a golden master, which would be used to stamp out a jintillion copies in preparation for its thunderous release a few weeks later. This news was received peevishly by the staff of Electric Communities, including one whose office door was plastered with the usual assortment of cartoons and novelties, e.g.

(2) a copy of a Dilbert cartoon in which Dilbert, the long-suffering corporate software engineer, encounters a portly, bearded, hairy man of a certain age–a bit like Santa Claus, but darker, with a certain edge about him. Dilbert recognizes this man, based upon his appearance and affect, as a Unix hacker, and reacts with a certain mixture of nervousness, awe, and hostility. Dilbert jabs weakly at the disturbing interloper for a couple of frames; the Unix hacker listens with a kind of infuriating, beatific calm, then, in the last frame, reaches into his pocket. "Here's a nickel, kid," he says, "go buy yourself a real computer."

(3) the owner of the door, and the cartoon, was one Doug Barnes. Barnes was known to harbor certain heretical opinions on the subject of operating systems. Unlike most Bay Area techies who revered the Macintosh, considering it to be a true hacker's machine, Barnes was fond of pointing out that the Mac, with its hermetically sealed architecture, was actually hostile to hackers, who are prone to tinkering and dogmatic about openness. By

contrast, the IBM-compatible line of machines, which can easily be taken apart and plugged back together, was much more hackable.

So when I got home I began messing around with Linux, which is one of many, many different concrete implementations of the abstract, Platonic ideal called Unix. I was not looking forward to changing over to a new OS, because my credit cards were still smoking from all the money I'd spent on Mac hardware over the years. But Linux's great virtue was, and is, that it would run on exactly the same sort of hardware as the Microsoft OSes–which is to say, the cheapest hardware in existence. As if to demonstrate why this was a great idea, I was, within a week or two of returning home, able to get my hand on a then-decent computer (a 33-MHz 486 box) for free, because I knew a guy who worked in an office where they were simply being thrown away. Once I got it home, I yanked the hood off, stuck my hands in, and began switching cards around. If something didn't work, I went to a used-computer outlet and pawed through a bin full of components and bought a new card for a few bucks.

The availability of all this cheap but effective hardware was an unintended consequence of decisions that had been made more than a decade earlier by IBM and Microsoft. When Windows came out, and brought the GUI to a much larger market, the hardware regime changed: the cost of color video cards and high-resolution monitors began to drop, and is dropping still. This free-for-all approach to hardware meant that Windows was unavoidably clunky compared to MacOS. But the GUI brought computing to such a vast audience that volume went way up and prices collapsed. Meanwhile Apple, which so badly wanted a clean, integrated OS with video neatly integrated into processing hardware, had fallen far behind in market share, at least partly because their beautiful hardware cost so much.

But the price that we Mac owners had to pay for superior aesthetics and engineering was not merely a financial one. There was a cultural price too, stemming from the fact that we couldn't open up the hood and mess around with it. Doug Barnes was right. Apple, in spite of its reputation as the machine of choice of scruffy, creative hacker types, had actually created a machine that discouraged hacking, while Microsoft, viewed as a technological laggard and copycat, had created a vast, disorderly parts bazaar–a primordial soup that eventually self-assembled into Linux.

THE HOLE HAWG OF OPERATING SYSTEMS

Unix has always lurked provocatively in the background of the operating system wars, like the Russian Army. Most people know it only by reputation, and its reputation, as the Dilbert cartoon suggests, is mixed. But everyone seems to agree that if it could only get its act together and stop surrendering vast tracts of rich agricultural land and hundreds of thousands of prisoners of war to the onrushing invaders, it could stomp them (and all other opposition) flat.

It is difficult to explain how Unix has earned this respect without going into

mind-smashing technical detail. Perhaps the gist of it can be explained by telling a story about drills.

The Hole Hawg is a drill made by the Milwaukee Tool Company. If you look in a typical hardware store you may find smaller Milwaukee drills but not the Hole Hawg, which is too powerful and too expensive for homeowners. The Hole Hawg does not have the pistol-like design of a cheap homeowner's drill. It is a cube of solid metal with a handle sticking out of one face and a chuck mounted in another. The cube contains a disconcertingly potent electric motor. You can hold the handle and operate the trigger with your index finger, but unless you are exceptionally strong you cannot control the weight of the Hole Hawg with one hand; it is a two-hander all the way. In order to fight off the counter-torque of the Hole Hawg you use a separate handle (provided), which you screw into one side of the iron cube or the other depending on whether you are using your left or right hand to operate the trigger. This handle is not a sleek, ergonomically designed item as it would be in a homeowner's drill. It is simply a foot-long chunk of regular galvanized pipe, threaded on one end, with a black rubber handle on the other. If you lose it, you just go to the local plumbing supply store and buy another chunk of pipe.

During the Eighties I did some construction work. One day, another worker leaned a ladder against the outside of the building that we were putting up, climbed up to the second-story level, and used the Hole Hawg to drill a hole through the exterior wall. At some point, the drill bit caught in the wall. The Hole Hawg, following its one and only imperative, kept going. It spun the worker's body around like a rag doll, causing him to knock his own ladder down. Fortunately he kept his grip on the Hole Hawg, which remained lodged in the wall, and he simply dangled from it and shouted for help until someone came along and reinstated the ladder.

I myself used a Hole Hawg to drill many holes through studs, which it did as a blender chops cabbage. I also used it to cut a few six-inch-diameter holes through an old lath-and-plaster ceiling. I chucked in a new hole saw, went up to the second story, reached down between the newly installed floor joists, and began to cut through the first-floor ceiling below. Where my homeowner's drill had labored and whined to spin the huge bit around, and had stalled at the slightest obstruction, the Hole Hawg rotated with the stupid consistency of a spinning planet. When the hole saw seized up, the Hole Hawg spun itself and me around, and crushed one of my hands between the steel pipe handle and a joist, producing a few lacerations, each surrounded by a wide corona of deeply bruised flesh. It also bent the hole saw itself, though not so badly that I couldn't use it. After a few such run-ins, when I got ready to use the Hole Hawg my heart actually began to pound with atavistic terror.

But I never blamed the Hole Hawg; I blamed myself. The Hole Hawg is dangerous because it does exactly what you tell it to. It is not bound by the physical limitations that are inherent in a cheap drill, and neither is it limited by safety interlocks that might be built into a homeowner's product by a liability-conscious

manufacturer. The danger lies not in the machine itself but in the user's failure to envision the full consequences of the instructions he gives to it.

A smaller tool is dangerous too, but for a completely different reason: it tries to do what you tell it to, and fails in some way that is unpredictable and almost always undesirable. But the Hole Hawg is like the genie of the ancient fairy tales, who carries out his master's instructions literally and precisely and with unlimited power, often with disastrous, unforeseen consequences.

Pre-Hole Hawg, I used to examine the drill selection in hardware stores with what I thought was a judicious eye, scorning the smaller low-end models and hefting the big expensive ones appreciatively, wishing I could afford one of them babies. Now I view them all with such contempt that I do not even consider them to be real drills–merely scaled-up toys designed to exploit the self-delusional tendencies of soft-handed homeowners who want to believe that they have purchased an actual tool. Their plastic casings, carefully designed and focus-group-tested to convey a feeling of solidity and power, seem disgustingly flimsy and cheap to me, and I am ashamed that I was ever bamboozled into buying such knicknacks.

It is not hard to imagine what the world would look like to someone who had been raised by contractors and who had never used any drill other than a Hole Hawg. Such a person, presented with the best and most expensive hardware-store drill, would not even recognize it as such. He might instead misidentify it as a child's toy, or some kind of motorized screwdriver. If a salesperson or a deluded homeowner referred to it as a drill, he would laugh and tell them that they were mistaken–they simply had their terminology wrong. His interlocutor would go away irritated, and probably feeling rather defensive about his basement full of cheap, dangerous, flashy, colorful tools.

Unix is the Hole Hawg of operating systems, and Unix hackers, like Doug Barnes and the guy in the Dilbert cartoon and many of the other people who populate Silicon Valley, are like contractor's sons who grew up using only Hole Hawgs. They might use Apple/Microsoft OSes to write letters, play video games, or balance their checkbooks, but they cannot really bring themselves to take these operating systems seriously.

THE ORAL TRADITION

Unix is hard to learn. The process of learning it is one of multiple small epiphanies. Typically you are just on the verge of inventing some necessary tool or utility when you realize that someone else has already invented it, and built it in, and this explains some odd file or directory or command that you have noticed but never really understood before.

For example there is a command (a small program, part of the OS) called whoami, which enables you to ask the computer who it thinks you are. On a Unix machine, you are always logged in under some name–possibly even your own! What files you may work with, and what software you may use, depends on your identity. When I started out using Linux, I was on a non-networked

machine in my basement, with only one user account, and so when I became aware of the whoami command it struck me as ludicrous. But once you are logged in as one person, you can temporarily switch over to a pseudonym in order to access different files. If your machine is on the Internet, you can log onto other computers, provided you have a user name and a password. At that point the distant machine becomes no different in practice from the one right in front of you. These changes in identity and location can easily become nested inside each other, many layers deep, even if you aren't doing anything nefarious. Once you have forgotten who and where you are, the whoami command is indispensible. I use it all the time.

The file systems of Unix machines all have the same general structure. On your flimsy operating systems, you can create directories (folders) and give them names like Frodo or My Stuff and put them pretty much anywhere you like. But under Unix the highest level–the root–of the filesystem is always designated with the single character "/" and it always contains the same set of top-level directories:

/usr /etc /var /bin /proc /boot /home /root /sbin /dev /lib /tmp

and each of these directories typically has its own distinct structure of subdirectories. Note the obsessive use of abbreviations and avoidance of capital letters; this is a system invented by people to whom repetitive stress disorder is what black lung is to miners. Long names get worn down to three-letter nubbins, like stones smoothed by a river.

This is not the place to try to explain why each of the above directories exists, and what is contained in it. At first it all seems obscure; worse, it seems deliberately obscure. When I started using Linux I was accustomed to being able to create directories wherever I wanted and to give them whatever names struck my fancy. Under Unix you are free to do that, of course (you are free to do anything) but as you gain experience with the system you come to understand that the directories listed above were created for the best of reasons and that your life will be much easier if you follow along (within /home, by the way, you have pretty much unlimited freedom).

After this kind of thing has happened several hundred or thousand times, the hacker understands why Unix is the way it is, and agrees that it wouldn't be the same any other way. It is this sort of acculturation that gives Unix hackers their confidence in the system, and the attitude of calm, unshakable, annoying superiority captured in the Dilbert cartoon. Windows 95 and MacOS are products, contrived by engineers in the service of specific companies. Unix, by contrast, is not so much a product as it is a painstakingly compiled oral history of the hacker subculture. It is our Gilgamesh epic.

What made old epics like Gilgamesh so powerful and so long-lived was that they were living bodies of narrative that many people knew by heart, and told over and over again–making their own personal embellishments whenever it struck their fancy. The bad embellishments were shouted down, the good ones picked

up by others, polished, improved, and, over time, incorporated into the story. Likewise, Unix is known, loved, and understood by so many hackers that it can be re-created from scratch whenever someone needs it. This is very difficult to understand for people who are accustomed to thinking of OSes as things that absolutely have to be bought.

Many hackers have launched more or less successful re-implementations of the Unix ideal. Each one brings in new embellishments. Some of them die out quickly, some are merged with similar, parallel innovations created by different hackers attacking the same problem, others still are embraced, and adopted into the epic. Thus Unix has slowly accreted around a simple kernel and acquired a kind of complexity and asymmetry about it that is organic, like the roots of a tree, or the branchings of a coronary artery. Understanding it is more like anatomy than physics.

For at least a year, prior to my adoption of Linux, I had been hearing about it. Credible, well-informed people kept telling me that a bunch of hackers had got together an implentation of Unix that could be downloaded, free of charge, from the Internet. For a long time I could not bring myself to take the notion seriously. It was like hearing rumors that a group of model rocket enthusiasts had created a completely functional Saturn V by exchanging blueprints on the Net and mailing valves and flanges to each other.

But it's true. Credit for Linux generally goes to its human namesake, one Linus Torvalds, a Finn who got the whole thing rolling in 1991 when he used some of the GNU tools to write the beginnings of a Unix kernel that could run on PC-compatible hardware. And indeed Torvalds deserves all the credit he has ever gotten, and a whole lot more. But he could not have made it happen by himself, any more than Richard Stallman could have. To write code at all, Torvalds had to have cheap but powerful development tools, and these he got from Stallman's GNU project.

And he had to have cheap hardware on which to write that code. Cheap hardware is a much harder thing to arrange than cheap software; a single person (Stallman) can write software and put it up on the Net for free, but in order to make hardware it's necessary to have a whole industrial infrastructure, which is not cheap by any stretch of the imagination. Really the only way to make hardware cheap is to punch out an incredible number of copies of it, so that the unit cost eventually drops. For reasons already explained, Apple had no desire to see the cost of hardware drop. The only reason Torvalds had cheap hardware was Microsoft.

Microsoft refused to go into the hardware business, insisted on making its software run on hardware that anyone could build, and thereby created the market conditions that allowed hardware prices to plummet. In trying to understand the Linux phenomenon, then, we have to look not to a single innovator but to a sort of bizarre Trinity: Linus Torvalds, Richard Stallman, and Bill Gates. Take away any of these three and Linux would not exist.

OS SHOCK

Young Americans who leave their great big homogeneous country and visit some other part of the world typically go through several stages of culture shock: first, dumb wide-eyed astonishment. Then a tentative engagement with the new country's manners, cuisine, public transit systems and toilets, leading to a brief period of fatuous confidence that they are instant experts on the new country. As the visit wears on, homesickness begins to set in, and the traveler begins to appreciate, for the first time, how much he or she took for granted at home. At the same time it begins to seem obvious that many of one's own cultures and traditions are essentially arbitrary, and could have been different; driving on the right side of the road, for example. When the traveler returns home and takes stock of the experience, he or she may have learned a good deal more about America than about the country they went to visit.

For the same reasons, Linux is worth trying. It is a strange country indeed, but you don't have to live there; a brief sojourn suffices to give some flavor of the place and–more importantly–to lay bare everything that is taken for granted, and all that could have been done differently, under Windows or MacOS.

You can't try it unless you install it. With any other OS, installing it would be a straightforward transaction: in exchange for money, some company would give you a CD-ROM, and you would be on your way. But a lot is subsumed in that kind of transaction, and has to be gone through and picked apart.

We like plain dealings and straightforward transactions in America. If you go to Egypt and, say, take a taxi somewhere, you become a part of the taxi driver's life; he refuses to take your money because it would demean your friendship, he follows you around town, and weeps hot tears when you get in some other guy's taxi. You end up meeting his kids at some point, and have to devote all sort of ingenuity to finding some way to compensate him without insulting his honor. It is exhausting. Sometimes you just want a simple Manhattan-style taxi ride.

But in order to have an American-style setup, where you can just go out and hail a taxi and be on your way, there must exist a whole hidden apparatus of medallions, inspectors, commissions, and so forth–which is fine as long as taxis are cheap and you can always get one. When the system fails to work in some way, it is mysterious and infuriating and turns otherwise reasonable people into conspiracy theorists. But when the Egyptian system breaks down, it breaks down transparently. You can't get a taxi, but your driver's nephew will show up, on foot, to explain the problem and apologize.

Microsoft and Apple do things the Manhattan way, with vast complexity hidden behind a wall of interface. Linux does things the Egypt way, with vast complexity strewn about all over the landscape. If you've just flown in from Manhattan, your first impulse will be to throw up your hands and say "For crying out loud! Will you people get a grip on yourselves!?" But this does not make friends in Linux-land any better than it would in Egypt.

You can suck Linux right out of the air, as it were, by downloading the right files and putting them in the right places, but there probably are not more than a few hundred people in the world who could create a functioning Linux system in that way. What you really need is a distribution of Linux, which means a prepackaged set of files. But distributions are a separate thing from Linux per se.

Linux per se is not a specific set of ones and zeroes, but a self-organizing Net subculture. The end result of its collective lucubrations is a vast body of source code, almost all written in C (the dominant computer programming language). "Source code" just means a computer program as typed in and edited by some hacker. If it's in C, the file name will probably have .c or .cpp on the end of it, depending on which dialect was used; if it's in some other language it will have some other suffix. Frequently these sorts of files can be found in a directory with the name /src which is the hacker's Hebraic abbreviation of "source."

Source files are useless to your computer, and of little interest to most users, but they are of gigantic cultural and political significance, because Microsoft and Apple keep them secret while Linux makes them public. They are the family jewels. They are the sort of thing that in Hollywood thrillers is used as a McGuffin: the plutonium bomb core, the top-secret blueprints, the suitcase of bearer bonds, the reel of microfilm. If the source files for Windows or MacOS were made public on the Net, then those OSes would become free, like Linux–only not as good, because no one would be around to fix bugs and answer questions. Linux is "open source" software meaning, simply, that anyone can get copies of its source code files.

Your computer doesn't want source code any more than you do; it wants object code. Object code files typically have the suffix .o and are unreadable all but a few, highly strange humans, because they consist of ones and zeroes. Accordingly, this sort of file commonly shows up in a directory with the name /bin, for "binary."

Source files are simply ASCII text files. ASCII denotes a particular way of encoding letters into bit patterns. In an ASCII file, each character has eight bits all to itself. This creates a potential "alphabet" of 256 distinct characters, in that eight binary digits can form that many unique patterns. In practice, of course, we tend to limit ourselves to the familiar letters and digits. The bit-patterns used to represent those letters and digits are the same ones that were physically punched into the paper tape by my high school teletype, which in turn were the same one used by the telegraph industry for decades previously. ASCII text files, in other words, are telegrams, and as such they have no typographical frills. But for the same reason they are eternal, because the code never changes, and universal, because every text editing and word processing software ever written knows about this code.

Therefore just about any software can be used to create, edit, and read source code files. Object code files, then, are created from these source files by a piece

of software called a compiler, and forged into a working application by another piece of software called a linker.

The triad of editor, compiler, and linker, taken together, form the core of a software development system. Now, it is possible to spend a lot of money on shrink-wrapped development systems with lovely graphical user interfaces and various ergonomic enhancements. In some cases it might even be a good and reasonable way to spend money. But on this side of the road, as it were, the very best software is usually the free stuff. Editor, compiler and linker are to hackers what ponies, stirrups, and archery sets were to the Mongols. Hackers live in the saddle, and hack on their own tools even while they are using them to create new applications. It is quite inconceivable that superior hacking tools could have been created from a blank sheet of paper by product engineers. Even if they are the brightest engineers in the world they are simply outnumbered.

In the GNU/Linux world there are two major text editing programs: the minimalist vi (known in some implementations as elvis) and the maximalist emacs. I use emacs, which might be thought of as a thermonuclear word processor. It was created by Richard Stallman; enough said. It is written in Lisp, which is the only computer language that is beautiful. It is colossal, and yet it only edits straight ASCII text files, which is to say, no fonts, no boldface, no underlining. In other words, the engineer-hours that, in the case of Microsoft Word, were devoted to features like mail merge, and the ability to embed feature-length motion pictures in corporate memoranda, were, in the case of emacs, focused with maniacal intensity on the deceptively simple-seeming problem of editing text. If you are a professional writer–i.e., if someone else is getting paid to worry about how your words are formatted and printed–emacs outshines all other editing software in approximately the same way that the noonday sun does the stars. It is not just bigger and brighter; it simply makes everything else vanish. For page layout and printing you can use TeX: a vast corpus of typesetting lore written in C and also available on the Net for free.

I could say a lot about emacs and TeX, but right now I am trying to tell a story about how to actually install Linux on your machine. The hard-core survivalist approach would be to download an editor like emacs, and the GNU Tools–the compiler and linker–which are polished and excellent to the same degree as emacs. Equipped with these, one would be able to start downloading ASCII source code files (/src) and compiling them into binary object code files (/bin) that would run on the machine. But in order to even arrive at this point–to get emacs running, for example–you have to have Linux actually up and running on your machine. And even a minimal Linux operating system requires thousands of binary files all acting in concert, and arranged and linked together just so.

Several entities have therefore taken it upon themselves to create "distributions" of Linux. If I may extend the Egypt analogy slightly, these entities are a bit like tour guides who meet you at the airport, who speak your language, and who help guide you through the initial culture shock. If you are an Egyptian, of course, you see it the other way; tour guides exist to keep brutish outlanders

from traipsing through your mosques and asking you the same questions over and over and over again.

Some of these tour guides are commercial organizations, such as Red Hat Software, which makes a Linux distribution called Red Hat that has a relatively commercial sheen to it. In most cases you put a Red Hat CD-ROM into your PC and reboot and it handles the rest. Just as a tour guide in Egypt will expect some sort of compensation for his services, commercial distributions need to be paid for. In most cases they cost almost nothing and are well worth it.

I use a distribution called Debian (the word is a contraction of "Deborah" and "Ian") which is non-commercial. It is organized (or perhaps I should say "it has organized itself") along the same lines as Linux in general, which is to say that it consists of volunteers who collaborate over the Net, each responsible for looking after a different chunk of the system. These people have broken Linux down into a number of packages, which are compressed files that can be downloaded to an already functioning Debian Linux system, then opened up and unpacked using a free installer application. Of course, as such, Debian has no commercial arm–no distribution mechanism. You can download all Debian packages over the Net, but most people will want to have them on a CD-ROM. Several different companies have taken it upon themselves to decoct all of the current Debian packages onto CD-ROMs and then sell them. I buy mine from Linux Systems Labs. The cost for a three-disc set, containing Debian in its entirety, is less than three dollars. But (and this is an important distinction) not a single penny of that three dollars is going to any of the coders who created Linux, nor to the Debian packagers. It goes to Linux Systems Labs and it pays, not for the software, or the packages, but for the cost of stamping out the CD-ROMs.

Every Linux distribution embodies some more or less clever hack for circumventing the normal boot process and causing your computer, when it is turned on, to organize itself, not as a PC running Windows, but as a "host" running Unix. This is slightly alarming the first time you see it, but completely harmless. When a PC boots up, it goes through a little self-test routine, taking an inventory of available disks and memory, and then begins looking around for a disk to boot up from. In any normal Windows computer that disk will be a hard drive. But if you have your system configured right, it will look first for a floppy or CD-ROM disk, and boot from that if one is available.

Linux exploits this chink in the defenses. Your computer notices a bootable disk in the floppy or CD-ROM drive, loads in some object code from that disk, and blindly begins to execute it. But this is not Microsoft or Apple code, this is Linux code, and so at this point your computer begins to behave very differently from what you are accustomed to. Cryptic messages began to scroll up the screen. If you had booted a commercial OS, you would, at this point, be seeing a "Welcome to MacOS" cartoon, or a screen filled with clouds in a blue sky, and a Windows logo. But under Linux you get a long telegram printed in stark white letters on a black screen. There is no "welcome!" message. Most of the telegram has the semi-inscrutable menace of graffiti tags.

Dec 14 15:04:15 theRev syslogd 1.3-3#17: restart. Dec 14 15:04:15 theRev kernel: klogd 1.3-3, log source = /proc/kmsg started. Dec 14 15:04:15 theRev kernel: Loaded 3535 symbols from /System.map. Dec 14 15:04:15 theRev kernel: Symbols match kernel version 2.0.30. Dec 14 15:04:15 theRev kernel: No module symbols loaded. Dec 14 15:04:15 theRev kernel: Intel MultiProcessor Specification v1.4 Dec 14 15:04:15 theRev kernel: Virtual Wire compatibility mode. Dec 14 15:04:15 theRev kernel: OEM ID: INTEL Product ID: 440FX APIC at: 0xFEE00000 Dec 14 15:04:15 theRev kernel: Processor #0 Pentium(tm) Pro APIC version 17 Dec 14 15:04:15 theRev kernel: Processor #1 Pentium(tm) Pro APIC version 17 Dec 14 15:04:15 theRev kernel: I/O APIC #2 Version 17 at 0xFEC00000. Dec 14 15:04:15 theRev kernel: Processors: 2 Dec 14 15:04:15 theRev kernel: Console: 16 point font, 400 scans Dec 14 15:04:15 theRev kernel: Console: colour VGA+ 80x25, 1 virtual console (max 63) Dec 14 15:04:15 theRev kernel: pcibios_init : BIOS32 Service Directory structure at 0x000fdb70 Dec 14 15:04:15 theRev kernel: pcibios_init : BIOS32 Service Directory entry at 0xfdb80 Dec 14 15:04:15 theRev kernel: pcibios_init : PCI BIOS revision 2.10 entry at 0xfdba1 Dec 14 15:04:15 theRev kernel: Probing PCI hardware. Dec 14 15:04:15 theRev kernel: Warning : Unknown PCI device (10b7:9001). Please read include/linux/pci.h Dec 14 15:04:15 theRev kernel: Calibrating delay loop.. ok - 179.40 BogoMIPS Dec 14 15:04:15 theRev kernel: Memory: 64268k/66556k available (700k kernel code, 384k reserved, 1204k data) Dec 14 15:04:15 theRev kernel: Swansea University Computer Society NET3.035 for Linux 2.0 Dec 14 15:04:15 theRev kernel: NET3: Unix domain sockets 0.13 for Linux NET3.035. Dec 14 15:04:15 theRev kernel: Swansea University Computer Society TCP/IP for NET3.034 Dec 14 15:04:15 theRev kernel: IP Protocols: ICMP, UDP, TCP Dec 14 15:04:15 theRev kernel: Checking 386/387 coupling... Ok, fpu using exception 16 error reporting. Dec 14 15:04:15 theRev kernel: Checking 'hlt' instruction... Ok. Dec 14 15:04:15 theRev kernel: Linux version 2.0.30 (root@theRev) (gcc version 2.7.2.1) #15 Fri Mar 27 16:37:24 PST 1998 Dec 14 15:04:15 theRev kernel: Booting processor 1 stack 00002000: Calibrating delay loop.. ok - 179.40 BogoMIPS Dec 14 15:04:15 theRev kernel: Total of 2 processors activated (358.81 BogoMIPS). Dec 14 15:04:15 theRev kernel: Serial driver version 4.13 with no serial options enabled Dec 14 15:04:15 theRev kernel: tty00 at 0x03f8 (irq = 4) is a 16550A Dec 14 15:04:15 theRev kernel: tty01 at 0x02f8 (irq = 3) is a 16550A Dec 14 15:04:15 theRev kernel: lp1 at 0x0378, (polling) Dec 14 15:04:15 theRev kernel: PS/2 auxiliary pointing device detected – driver installed. Dec 14 15:04:15 theRev kernel: Real Time Clock Driver v1.07 Dec 14 15:04:15 theRev kernel: loop: registered device at major 7 Dec 14 15:04:15 theRev kernel: ide: i82371 PIIX (Triton) on PCI bus 0 function 57 Dec 14 15:04:15 theRev kernel: ide0: BM-DMA at 0xffa0-0xffa7 Dec 14 15:04:15 theRev kernel: ide1: BM-DMA at 0xffa8-0xffaf Dec 14 15:04:15 theRev kernel: hda: Conner Peripherals 1275MB - CFS1275A, 1219MB w/64kB Cache, LBA, CHS=619/64/63 Dec 14 15:04:15 theRev kernel: hdb: Maxtor 84320A5, 4119MB w/256kB Cache, LBA, CHS=8928/15/63, DMA Dec 14 15:04:15 theRev kernel: hdc: , ATAPI CDROM drive Dec 15 11:58:06 theRev kernel: ide0 at 0x1f0-0x1f7,0x3f6 on irq 14 Dec 15 11:58:06 theRev kernel: ide1 at 0x170-0x177,0x376

on irq 15 Dec 15 11:58:06 theRev kernel: Floppy drive(s): fd0 is 1.44M Dec 15 11:58:06 theRev kernel: Started kswapd v 1.4.2.2 Dec 15 11:58:06 theRev kernel: FDC 0 is a National Semiconductor PC87306 Dec 15 11:58:06 theRev kernel: md driver 0.35 MAX_MD_DEV=4, MAX_REAL=8 Dec 15 11:58:06 theRev kernel: PPP: version 2.2.0 (dynamic channel allocation) Dec 15 11:58:06 theRev kernel: TCP compression code copyright 1989 Regents of the University of California Dec 15 11:58:06 theRev kernel: PPP Dynamic channel allocation code copyright 1995 Caldera, Inc. Dec 15 11:58:06 theRev kernel: PPP line discipline registered. Dec 15 11:58:06 theRev kernel: SLIP: version 0.8.4-NET3.019-NEWTTY (dynamic channels, max=256). Dec 15 11:58:06 theRev kernel: eth0: 3Com 3c900 Boomerang 10Mbps/Combo at 0xef00, 00:60:08:a4:3c:db, IRQ 10 Dec 15 11:58:06 theRev kernel: 8K word-wide RAM 3:5 Rx:Tx split, 10base2 interface. Dec 15 11:58:06 theRev kernel: Enabling bus-master transmits and whole-frame receives. Dec 15 11:58:06 theRev kernel: 3c59x.c:v0.49 1/2/98 Donald Becker http://cesdis.gsfc.nasa.gov/linux/drivers/vortex.html Dec 15 11:58:06 theRev kernel: Partition check: Dec 15 11:58:06 theRev kernel: hda: hda1 hda2 hda3 Dec 15 11:58:06 theRev kernel: hdb: hdb1 hdb2 Dec 15 11:58:06 theRev kernel: VFS: Mounted root (ext2 filesystem) readonly. Dec 15 11:58:06 theRev kernel: Adding Swap: 16124k swap-space (priority -1) Dec 15 11:58:06 theRev kernel: EXT2-fs warning: maximal mount count reached, running e2fsck is recommended Dec 15 11:58:06 theRev kernel: hdc: media changed Dec 15 11:58:06 theRev kernel: ISO9660 Extensions: RRIP_1991A Dec 15 11:58:07 theRev syslogd 1.3-3#17: restart. Dec 15 11:58:09 theRev diald[87]: Unable to open options file /etc/diald/diald.options: No such file or directory Dec 15 11:58:09 theRev diald[87]: No device specified. You must have at least one device! Dec 15 11:58:09 theRev diald[87]: You must define a connector script (option 'connect'). Dec 15 11:58:09 theRev diald[87]: You must define the remote ip address. Dec 15 11:58:09 theRev diald[87]: You must define the local ip address. Dec 15 11:58:09 theRev diald[87]: Terminating due to damaged reconfigure.

The only parts of this that are readable, for normal people, are the error messages and warnings. And yet it's noteworthy that Linux doesn't stop, or crash, when it encounters an error; it spits out a pithy complaint, gives up on whatever processes were damaged, and keeps on rolling. This was decidedly not true of the early versions of Apple and Microsoft OSes, for the simple reason that an OS that is not capable of walking and chewing gum at the same time cannot possibly recover from errors. Looking for, and dealing with, errors requires a separate process running in parallel with the one that has erred. A kind of superego, if you will, that keeps an eye on all of the others, and jumps in when one goes astray. Now that MacOS and Windows can do more than one thing at a time they are much better at dealing with errors than they used to be, but they are not even close to Linux or other Unices in this respect; and their greater complexity has made them vulnerable to new types of errors.

FALLIBILITY, ATONEMENT, REDEMPTION, TRUST, AND OTHER ARCANE TECHNICAL CONCEPTS

Linux is not capable of having any centrally organized policies dictating how to write error messages and documentation, and so each programmer writes his own. Usually they are in English even though tons of Linux programmers are Europeans. Frequently they are funny. Always they are honest. If something bad has happened because the software simply isn't finished yet, or because the user screwed something up, this will be stated forthrightly. The command line interface makes it easy for programs to dribble out little comments, warnings, and messages here and there. Even if the application is imploding like a damaged submarine, it can still usually eke out a little S.O.S. message. Sometimes when you finish working with a program and shut it down, you find that it has left behind a series of mild warnings and low-grade error messages in the command-line interface window from which you launched it. As if the software were chatting to you about how it was doing the whole time you were working with it.

Documentation, under Linux, comes in the form of man (short for manual) pages. You can access these either through a GUI (xman) or from the command line (man). Here is a sample from the man page for a program called rsh:

"Stop signals stop the local rsh process only; this is arguably wrong, but currently hard to fix for reasons too complicated to explain here."

The man pages contain a lot of such material, which reads like the terse mutterings of pilots wrestling with the controls of damaged airplanes. The general feel is of a thousand monumental but obscure struggles seen in the stop-action light of a strobe. Each programmer is dealing with his own obstacles and bugs; he is too busy fixing them, and improving the software, to explain things at great length or to maintain elaborate pretensions.

In practice you hardly ever encounter a serious bug while running Linux. When you do, it is almost always with commercial software (several vendors sell software that runs under Linux). The operating system and its fundamental utility programs are too important to contain serious bugs. I have been running Linux every day since late 1995 and have seen many application programs go down in flames, but I have never seen the operating system crash. Never. Not once. There are quite a few Linux systems that have been running continuously and working hard for months or years without needing to be rebooted.

Commercial OSes have to adopt the same official stance towards errors as Communist countries had towards poverty. For doctrinal reasons it was not possible to admit that poverty was a serious problem in Communist countries, because the whole point of Communism was to eradicate poverty. Likewise, commercial OS companies like Apple and Microsoft can't go around admitting that their software has bugs and that it crashes all the time, any more than Disney can issue press releases stating that Mickey Mouse is an actor in a suit.

This is a problem, because errors do exist and bugs do happen. Every few months Bill Gates tries to demo a new Microsoft product in front of a large audience only to have it blow up in his face. Commercial OS vendors, as a direct consequence of being commercial, are forced to adopt the grossly disingenuous position that bugs

are rare aberrations, usually someone else's fault, and therefore not really worth talking about in any detail. This posture, which everyone knows to be absurd, is not limited to press releases and ad campaigns. It informs the whole way these companies do business and relate to their customers. If the documentation were properly written, it would mention bugs, errors, and crashes on every single page. If the on-line help systems that come with these OSes reflected the experiences and concerns of their users, they would largely be devoted to instructions on how to cope with crashes and errors.

But this does not happen. Joint stock corporations are wonderful inventions that have given us many excellent goods and services. They are good at many things. Admitting failure is not one of them. Hell, they can't even admit minor shortcomings.

Of course, this behavior is not as pathological in a corporation as it would be in a human being. Most people, nowadays, understand that corporate press releases are issued for the benefit of the corporation's shareholders and not for the enlightenment of the public. Sometimes the results of this institutional dishonesty can be dreadful, as with tobacco and asbestos. In the case of commercial OS vendors it is nothing of the kind, of course; it is merely annoying.

Some might argue that consumer annoyance, over time, builds up into a kind of hardened plaque that can conceal serious decay, and that honesty might therefore be the best policy in the long run; the jury is still out on this in the operating system market. The business is expanding fast enough that it's still much better to have billions of chronically annoyed customers than millions of happy ones.

Most system administrators I know who work with Windows NT all the time agree that when it hits a snag, it has to be re-booted, and when it gets seriously messed up, the only way to fix it is to re-install the operating system from scratch. Or at least this is the only way that they know of to fix it, which amounts to the same thing. It is quite possible that the engineers at Microsoft have all sorts of insider knowledge on how to fix the system when it goes awry, but if they do, they do not seem to be getting the message out to any of the actual system administrators I know.

Because Linux is not commercial–because it is, in fact, free, as well as rather difficult to obtain, install, and operate–it does not have to maintain any pretensions as to its reliability. Consequently, it is much more reliable. When something goes wrong with Linux, the error is noticed and loudly discussed right away. Anyone with the requisite technical knowledge can go straight to the source code and point out the source of the error, which is then rapidly fixed by whichever hacker has carved out responsibility for that particular program.

As far as I know, Debian is the only Linux distribution that has its own constitution (http://www.debian.org/devel/constitution), but what really sold me on it was its phenomenal bug database (http://www.debian.org/Bugs), which is a sort of interactive Doomsday Book of error, fallibility, and redemption. It is simplicity itself. When had a problem with Debian in early January of 1997,

I sent in a message describing the problem to submit@bugs.debian.org. My problem was promptly assigned a bug report number (#6518) and a severity level (the available choices being critical, grave, important, normal, fixed, and wishlist) and forwarded to mailing lists where Debian people hang out. Within twenty-four hours I had received five e-mails telling me how to fix the problem: two from North America, two from Europe, and one from Australia. All of these e-mails gave me the same suggestion, which worked, and made my problem go away. But at the same time, a transcript of this exchange was posted to Debian's bug database, so that if other users had the same problem later, they would be able to search through and find the solution without having to enter a new, redundant bug report.

Contrast this with the experience that I had when I tried to install Windows NT 4.0 on the very same machine about ten months later, in late 1997. The installation program simply stopped in the middle with no error messages. I went to the Microsoft Support website and tried to perform a search for existing help documents that would address my problem. The search engine was completely nonfunctional; it did nothing at all. It did not even give me a message telling me that it was not working.

Eventually I decided that my motherboard must be at fault; it was of a slightly unusual make and model, and NT did not support as many different motherboards as Linux. I am always looking for excuses, no matter how feeble, to buy new hardware, so I bought a new motherboard that was Windows NT logo-compatible, meaning that the Windows NT logo was printed right on the box. I installed this into my computer and got Linux running right away, then attempted to install Windows NT again. Again, the installation died without any error message or explanation. By this time a couple of weeks had gone by and I thought that perhaps the search engine on the Microsoft Support website might be up and running. I gave that a try but it still didn't work.

So I created a new Microsoft support account, then logged on to submit the incident. I supplied my product ID number when asked, and then began to follow the instructions on a series of help screens. In other words, I was submitting a bug report just as with the Debian bug tracking system. It's just that the interface was slicker–I was typing my complaint into little text-editing boxes on Web forms, doing it all through the GUI, whereas with Debian you send in an e-mail telegram. I knew that when I was finished submitting the bug report, it would become proprietary Microsoft information, and other users wouldn't be able to see it. Many Linux users would refuse to participate in such a scheme on ethical grounds, but I was willing to give it a shot as an experiment. In the end, though I was never able to submit my bug report, because the series of linked web pages that I was filling out eventually led me to a completely blank page: a dead end.

So I went back and clicked on the buttons for "phone support" and eventually was given a Microsoft telephone number. When I dialed this number I got a series of piercing beeps and a recorded message from the phone company saying

"We're sorry, your call cannot be completed as dialed."

I tried the search page again–it was still completely nonfunctional. Then I tried PPI (Pay Per Incident) again. This led me through another series of Web pages until I dead-ended at one reading: "Notice-there is no Web page matching your request."

I tried it again, and eventually got to a Pay Per Incident screen reading: "OUT OF INCIDENTS. There are no unused incidents left in your account. If you would like to purchase a support incident, click OK-you will then be able to prepay for an incident. . . ." The cost per incident was $95.

The experiment was beginning to seem rather expensive, so I gave up on the PPI approach and decided to have a go at the FAQs posted on Microsoft's website. None of the available FAQs had anything to do with my problem except for one entitled "I am having some problems installing NT" which appeared to have been written by flacks, not engineers.

So I gave up and still, to this day, have never gotten Windows NT installed on that particular machine. For me, the path of least resistance was simply to use Debian Linux.

In the world of open source software, bug reports are useful information. Making them public is a service to other users, and improves the OS. Making them public systematically is so important that highly intelligent people voluntarily put time and money into running bug databases. In the commercial OS world, however, reporting a bug is a privilege that you have to pay lots of money for. But if you pay for it, it follows that the bug report must be kept confidential–otherwise anyone could get the benefit of your ninety-five bucks! And yet nothing prevents NT users from setting up their own public bug database.

This is, in other words, another feature of the OS market that simply makes no sense unless you view it in the context of culture. What Microsoft is selling through Pay Per Incident isn't technical support so much as the continued illusion that its customers are engaging in some kind of rational business transaction. It is a sort of routine maintenance fee for the upkeep of the fantasy. If people really wanted a solid OS they would use Linux, and if they really wanted tech support they would find a way to get it; Microsoft's customers want something else.

As of this writing (Jan. 1999), something like 32,000 bugs have been reported to the Debian Linux bug database. Almost all of them have been fixed a long time ago. There are twelve "critical" bugs still outstanding, of which the oldest was posted 79 days ago. There are 20 outstanding "grave" bugs of which the oldest is 1166 days old. There are 48 "important" bugs and hundreds of "normal" and less important ones.

Likewise, BeOS (which I'll get to in a minute) has its own bug database (http://www.be.com/developers/bugs/index.html) with its own classification system, including such categories as "Not a Bug," "Acknowledged Feature," and "Will Not Fix." Some of the "bugs" here are nothing more than Be hackers

blowing off steam, and are classified as "Input Acknowledged." For example, I found one that was posted on December 30th, 1998. It's in the middle of a long list of bugs, wedged between one entitled "Mouse working in very strange fashion" and another called "Change of BView frame does not affect, if BView not attached to a BWindow."

This one is entitled

R4: BeOS missing megalomaniacal figurehead to harness and focus developer rage

and it goes like this:

Be Status: Input Acknowledged BeOS Version: R3.2 Component: unknown

Full Description:

The BeOS needs a megalomaniacal egomaniac sitting on its throne to give it a human character which everyone loves to hate. Without this, the BeOS will languish in the impersonifiable realm of OSs that people can never quite get a handle on. You can judge the success of an OS not by the quality of its features, but by how infamous and disliked the leaders behind them are.

I believe this is a side-effect of developer comraderie under miserable conditions. After all, misery loves company. I believe that making the BeOS less conceptually accessible and far less reliable will require developers to band together, thus developing the kind of community where strangers talk to one- another, kind of like at a grocery store before a huge snowstorm.

Following this same program, it will likely be necessary to move the BeOS headquarters to a far-less-comfortable climate. General environmental discomfort will breed this attitude within and there truly is no greater recipe for success. I would suggest Seattle, but I think it's already taken. You might try Washington, DC, but definitely not somewhere like San Diego or Tucson.

Unfortunately, the Be bug reporting system strips off the names of the people who report the bugs (to protect them from retribution!?) and so I don't know who wrote this.

So it would appear that I'm in the middle of crowing about the technical and moral superiority of Debian Linux. But as almost always happens in the OS world, it's more complicated than that. I have Windows NT running on another machine, and the other day (Jan. 1999), when I had a problem with it, I decided to have another go at Microsoft Support. This time the search engine actually worked (though in order to reach it I had to identify myself as "advanced").

And instead of coughing up some useless FAQ, it located about two hundred documents (I was using very vague search criteria) that were obviously bug reports–though they were called something else. Microsoft, in other words, has got a system up and running that is functionally equivalent to Debian's bug database. It looks and feels different, of course, but it contains technical nitty-gritty and makes no bones about the existence of errors.

As I've explained, selling OSes for money is a basically untenable position, and the only way Apple and Microsoft can get away with it is by pursuing technological advancements as aggressively as they can, and by getting people to believe in, and to pay for, a particular image: in the case of Apple, that of the creative free thinker, and in the case of Microsoft, that of the respectable techno-bourgeois. Just like Disney, they're making money from selling an interface, a magic mirror. It has to be polished and seamless or else the whole illusion is ruined and the business plan vanishes like a mirage.

Accordingly, it was the case until recently that the people who wrote manuals and created customer support websites for commercial OSes seemed to have been barred, by their employers' legal or PR departments, from admitting, even obliquely, that the software might contain bugs or that the interface might be suffering from the blinking twelve problem. They couldn't address users' actual difficulties. The manuals and websites were therefore useless, and caused even technically self-assured users to wonder whether they were going subtly insane.

When Apple engages in this sort of corporate behavior, one wants to believe that they are really trying their best. We all want to give Apple the benefit of the doubt, because mean old Bill Gates kicked the crap out of them, and because they have good PR. But when Microsoft does it, one almost cannot help becoming a paranoid conspiracist. Obviously they are hiding something from us! And yet they are so powerful! They are trying to drive us crazy!

This approach to dealing with one's customers was straight out of the Central European totalitarianism of the mid-Twentieth Century. The adjectives "Kafkaesque" and "Orwellian" come to mind. It couldn't last, any more than the Berlin Wall could, and so now Microsoft has a publicly available bug database. It's called something else, and it takes a while to find it, but it's there.

They have, in other words, adapted to the two-tiered Eloi/Morlock structure of technological society. If you're an Eloi you install Windows, follow the instructions, hope for the best, and dumbly suffer when it breaks. If you're a Morlock you go to the website, tell it that you are "advanced," find the bug database, and get the truth straight from some anonymous Microsoft engineer.

But once Microsoft has taken this step, it raises the question, once again, of whether there is any point to being in the OS business at all. Customers might be willing to pay $95 to report a problem to Microsoft if, in return, they get some advice that no other user is getting. This has the useful side effect of keeping the users alienated from one another, which helps maintain the illusion that bugs are rare aberrations. But once the results of those bug reports become

openly available on the Microsoft website, everything changes. No one is going to cough up $95 to report a problem when chances are good that some other sucker will do it first, and that instructions on how to fix the bug will then show up, for free, on a public website. And as the size of the bug database grows, it eventually becomes an open admission, on Microsoft's part, that their OSes have just as many bugs as their competitors'. There is no shame in that; as I mentioned, Debian's bug database has logged 32,000 reports so far. But it puts Microsoft on an equal footing with the others and makes it a lot harder for their customers–who want to believe–to believe.

MEMENTO MORI

Once the Linux machine has finished spitting out its jargonic opening telegram, it prompts me to log in with a user name and a password. At this point the machine is still running the command line interface, with white letters on a black screen. There are no windows, menus, or buttons. It does not respond to the mouse; it doesn't even know that the mouse is there. It is still possible to run a lot of software at this point. Emacs, for example, exists in both a CLI and a GUI version (actually there are two GUI versions, reflecting some sort of doctrinal schism between Richard Stallman and some hackers who got fed up with him). The same is true of many other Unix programs. Many don't have a GUI at all, and many that do are capable of running from the command line.

Of course, since my computer only has one monitor screen, I can only see one command line, and so you might think that I could only interact with one program at a time. But if I hold down the Alt key and then hit the F2 function button at the top of my keyboard, I am presented with a fresh, blank, black screen with a login prompt at the top of it. I can log in here and start some other program, then hit Alt-F1 and go back to the first screen, which is still doing whatever it was when I left it. Or I can do Alt-F3 and log in to a third screen, or a fourth, or a fifth. On one of these screens I might be logged in as myself, on another as root (the system administrator), on yet another I might be logged on to some other computer over the Internet.

Each of these screens is called, in Unix-speak, a tty, which is an abbreviation for teletype. So when I use my Linux system in this way I am going right back to that small room at Ames High School where I first wrote code twenty-five years ago, except that a tty is quieter and faster than a teletype, and capable of running vastly superior software, such as emacs or the GNU development tools.

It is easy (easy by Unix, not Apple/Microsoft standards) to configure a Linux machine so that it will go directly into a GUI when you boot it up. This way, you never see a tty screen at all. I still have mine boot into the white-on-black teletype screen however, as a computational memento mori. It used to be fashionable for a writer to keep a human skull on his desk as a reminder that he was mortal, that all about him was vanity. The tty screen reminds me that the same thing is true of slick user interfaces.

The X Windows System, which is the GUI of Unix, has to be capable of running on hundreds of different video cards with different chipsets, amounts of onboard memory, and motherboard buses. Likewise, there are hundreds of different types of monitors on the new and used market, each with different specifications, and so there are probably upwards of a million different possible combinations of card and monitor. The only thing they all have in common is that they all work in VGA mode, which is the old command-line screen that you see for a few seconds when you launch Windows. So Linux always starts in VGA, with a teletype interface, because at first it has no idea what sort of hardware is attached to your computer. In order to get beyond the glass teletype and into the GUI, you have to tell Linux exactly what kinds of hardware you have. If you get it wrong, you'll get a blank screen at best, and at worst you might actually destroy your monitor by feeding it signals it can't handle.

When I started using Linux this had to be done by hand. I once spent the better part of a month trying to get an oddball monitor to work for me, and filled the better part of a composition book with increasingly desperate scrawled notes. Nowadays, most Linux distributions ship with a program that automatically scans the video card and self-configures the system, so getting X Windows up and running is nearly as easy as installing an Apple/Microsoft GUI. The crucial information goes into a file (an ASCII text file, naturally) called XF86Config, which is worth looking at even if your distribution creates it for you automatically. For most people it looks like meaningless cryptic incantations, which is the whole point of looking at it. An Apple/Microsoft system needs to have the same information in order to launch its GUI, but it's apt to be deeply hidden somewhere, and it's probably in a file that can't even be opened and read by a text editor. All of the important files that make Linux systems work are right out in the open. They are always ASCII text files, so you don't need special tools to read them. You can look at them any time you want, which is good, and you can mess them up and render your system totally dysfunctional, which is not so good.

At any rate, assuming that my XF86Config file is just so, I enter the command "startx" to launch the X Windows System. The screen blanks out for a minute, the monitor makes strange twitching noises, then reconstitutes itself as a blank gray desktop with a mouse cursor in the middle. At the same time it is launching a window manager. X Windows is pretty low-level software; it provides the infrastructure for a GUI, and it's a heavy industrial infrastructure. But it doesn't do windows. That's handled by another category of application that sits atop X Windows, called a window manager. Several of these are available, all free of course. The classic is twm (Tom's Window Manager) but there is a smaller and supposedly more efficient variant of it called fvwm, which is what I use. I have my eye on a completely different window manager called Enlightenment, which may be the hippest single technology product I have ever seen, in that (a) it is for Linux, (b) it is freeware, (c) it is being developed by a very small number of obsessed hackers, and (d) it looks amazingly cool; it is the sort of window manager that might show up in the backdrop of an Aliens movie.

Anyway, the window manager acts as an intermediary between X Windows and whatever software you want to use. It draws the window frames, menus, and so on, while the applications themselves draw the actual content in the windows. The applications might be of any sort: text editors, Web browsers, graphics packages, or utility programs, such as a clock or calculator. In other words, from this point on, you feel as if you have been shunted into a parallel universe that is quite similar to the familiar Apple or Microsoft one, but slightly and pervasively different. The premier graphics program under Apple/Microsoft is Adobe Photoshop, but under Linux it's something called The GIMP. Instead of the Microsoft Office Suite, you can buy something called ApplixWare. Many commercial software packages, such as Mathematica, Netscape Communicator, and Adobe Acrobat, are available in Linux versions, and depending on how you set up your window manager you can make them look and behave just as they would under MacOS or Windows.

But there is one type of window you'll see on Linux GUI that is rare or nonexistent under other OSes. These windows are called "xterm" and contain nothing but lines of text–this time, black text on a white background, though you can make them be different colors if you choose. Each xterm window is a separate command line interface–a tty in a window. So even when you are in full GUI mode, you can still talk to your Linux machine through a command-line interface.

There are many good pieces of Unix software that do not have GUIs at all. This might be because they were developed before X Windows was available, or because the people who wrote them did not want to suffer through all the hassle of creating a GUI, or because they simply do not need one. In any event, those programs can be invoked by typing their names into the command line of an xterm window. The whoami command, mentioned earlier, is a good example. There is another called wc ("word count") which simply returns the number of lines, words, and characters in a text file.

The ability to run these little utility programs on the command line is a great virtue of Unix, and one that is unlikely to be duplicated by pure GUI operating systems. The wc command, for example, is the sort of thing that is easy to write with a command line interface. It probably does not consist of more than a few lines of code, and a clever programmer could probably write it in a single line. In compiled form it takes up just a few bytes of disk space. But the code required to give the same program a graphical user interface would probably run into hundreds or even thousands of lines, depending on how fancy the programmer wanted to make it. Compiled into a runnable piece of software, it would have a large overhead of GUI code. It would be slow to launch and it would use up a lot of memory. This would simply not be worth the effort, and so "wc" would never be written as an independent program at all. Instead users would have to wait for a word count feature to appear in a commercial software package.

GUIs tend to impose a large overhead on every single piece of software, even the smallest, and this overhead completely changes the programming environment. Small utility programs are no longer worth writing. Their functions, instead,

tend to get swallowed up into omnibus software packages. As GUIs get more complex, and impose more and more overhead, this tendency becomes more pervasive, and the software packages grow ever more colossal; after a point they begin to merge with each other, as Microsoft Word and Excel and PowerPoint have merged into Microsoft Office: a stupendous software Wal-Mart sitting on the edge of a town filled with tiny shops that are all boarded up.

It is an unfair analogy, because when a tiny shop gets boarded up it means that some small shopkeeper has lost his business. Of course nothing of the kind happens when "wc" becomes subsumed into one of Microsoft Word's countless menu items. The only real drawback is a loss of flexibility for the user, but it is a loss that most customers obviously do not notice or care about. The most serious drawback to the Wal-Mart approach is that most users only want or need a tiny fraction of what is contained in these giant software packages. The remainder is clutter, dead weight. And yet the user in the next cubicle over will have completely different opinions as to what is useful and what isn't.

The other important thing to mention, here, is that Microsoft has included a genuinely cool feature in the Office package: a Basic programming package. Basic is the first computer language that I learned, back when I was using the paper tape and the teletype. By using the version of Basic that comes with Office you can write your own little utility programs that know how to interact with all of the little doohickeys, gewgaws, bells, and whistles in Office. Basic is easier to use than the languages typically employed in Unix command-line programming, and Office has reached many, many more people than the GNU tools. And so it is quite possible that this feature of Office will, in the end, spawn more hacking than GNU.

But now I'm talking about application software, not operating systems. And as I've said, Microsoft's application software tends to be very good stuff. I don't use it very much, because I am nowhere near their target market. If Microsoft ever makes a software package that I use and like, then it really will be time to dump their stock, because I am a market segment of one.

GEEK FATIGUE

Over the years that I've been working with Linux I have filled three and a half notebooks logging my experiences. I only begin writing things down when I'm doing something complicated, like setting up X Windows or fooling around with my Internet connection, and so these notebooks contain only the record of my struggles and frustrations. When things are going well for me, I'll work along happily for many months without jotting down a single note. So these notebooks make for pretty bleak reading. Changing anything under Linux is a matter of opening up various of those little ASCII text files and changing a word here and a character there, in ways that are extremely significant to how the system operates.

Many of the files that control how Linux operates are nothing more than command lines that became so long and complicated that not even Linux hackers could

type them correctly. When working with something as powerful as Linux, you can easily devote a full half-hour to engineering a single command line. For example, the "find" command, which searches your file system for files that match certain criteria, is fantastically powerful and general. Its "man" is eleven pages long, and these are pithy pages; you could easily expand them into a whole book. And if that is not complicated enough in and of itself, you can always pipe the output of one Unix command to the input of another, equally complicated one. The "pon" command, which is used to fire up a PPP connection to the Internet, requires so much detailed information that it is basically impossible to launch it entirely from the command line. Instead you abstract big chunks of its input into three or four different files. You need a dialing script, which is effectively a little program telling it how to dial the phone and respond to various events; an options file, which lists up to about sixty different options on how the PPP connection is to be set up; and a secrets file, giving information about your password.

Presumably there are godlike Unix hackers somewhere in the world who don't need to use these little scripts and options files as crutches, and who can simply pound out fantastically complex command lines without making typographical errors and without having to spend hours flipping through documentation. But I'm not one of them. Like almost all Linux users, I depend on having all of those details hidden away in thousands of little ASCII text files, which are in turn wedged into the recesses of the Unix filesystem. When I want to change something about the way my system works, I edit those files. I know that if I don't keep track of every little change I've made, I won't be able to get your system back in working order after I've gotten it all messed up. Keeping hand-written logs is tedious, not to mention kind of anachronistic. But it's necessary.

I probably could have saved myself a lot of headaches by doing business with a company called Cygnus Support, which exists to provide assistance to users of free software. But I didn't, because I wanted to see if I could do it myself. The answer turned out to be yes, but just barely. And there are many tweaks and optimizations that I could probably make in my system that I have never gotten around to attempting, partly because I get tired of being a Morlock some days, and partly because I am afraid of fouling up a system that generally works well.

Though Linux works for me and many other users, its sheer power and generality is its Achilles' heel. If you know what you are doing, you can buy a cheap PC from any computer store, throw away the Windows discs that come with it, turn it into a Linux system of mind-boggling complexity and power. You can hook it up to twelve other Linux boxes and make it into part of a parallel computer. You can configure it so that a hundred different people can be logged onto it at once over the Internet, via as many modem lines, Ethernet cards, TCP/IP sockets, and packet radio links. You can hang half a dozen different monitors off of it and play DOOM with someone in Australia while tracking communications satellites in orbit and controlling your house's lights and thermostats and streaming live

video from your web-cam and surfing the Net and designing circuit boards on the other screens. But the sheer power and complexity of the system–the qualities that make it so vastly technically superior to other OSes–sometimes make it seem too formidable for routine day-to-day use.

Sometimes, in other words, I just want to go to Disneyland.

The ideal OS for me would be one that had a well-designed GUI that was easy to set up and use, but that included terminal windows where I could revert to the command line interface, and run GNU software, when it made sense. A few years ago, Be Inc. invented exactly that OS. It is called the BeOS.

- # Shell Scripting Tutorial

Type	Web Page
URL	http://www.shellscript.sh/
Accessed	10/25/2016, 4:29:06 PM
Abstract	From the Shell Scripting Tutorial at http://shellscript.sh/
Date Added	10/25/2016, 4:29:06 PM
Modified	10/25/2016, 4:29:06 PM

Notes:

- # Shell Scripting

â€œShell Scripting Tutorial.â€ Accessed October 25, 2016. **http://www.shellscript.sh/.**

- ## Unix Shell Scripting Tutorial

Type	Web Page
URL	http://supportweb.cs.bham.ac.uk/docs/tutorials/docsystem/build/tutorials/unixscripting/unixscripting.html
Accessed	10/25/2016, 4:29:15 PM
Date Added	10/25/2016, 4:29:15 PM
Modified	10/25/2016, 4:29:15 PM

Notes:

- # Shell Scripting

â€œUnix Shell Scripting Tutorial.â€ Accessed October 25, 2016.
http://supportweb.cs.bham.ac.uk/docs/tutorials/docsystem/build/tutorials/unixscripting/unixscripting.html

- # LaTeX-a document-preparation system

Type	Book
Author	Leslie Lampor
Date Added	10/25/2016, 7:34:49 PM
Modified	10/25/2016, 7:35:18 PM

Notes:

- # LaTeX explained by its author

  ```
  pdftk LaTeX-a\ document-preparation\ system-2ed_La\ -\ Leslie\ Lampor.pdf cat 17-26 output Latex_Lampor_chp1.pdf
  ```

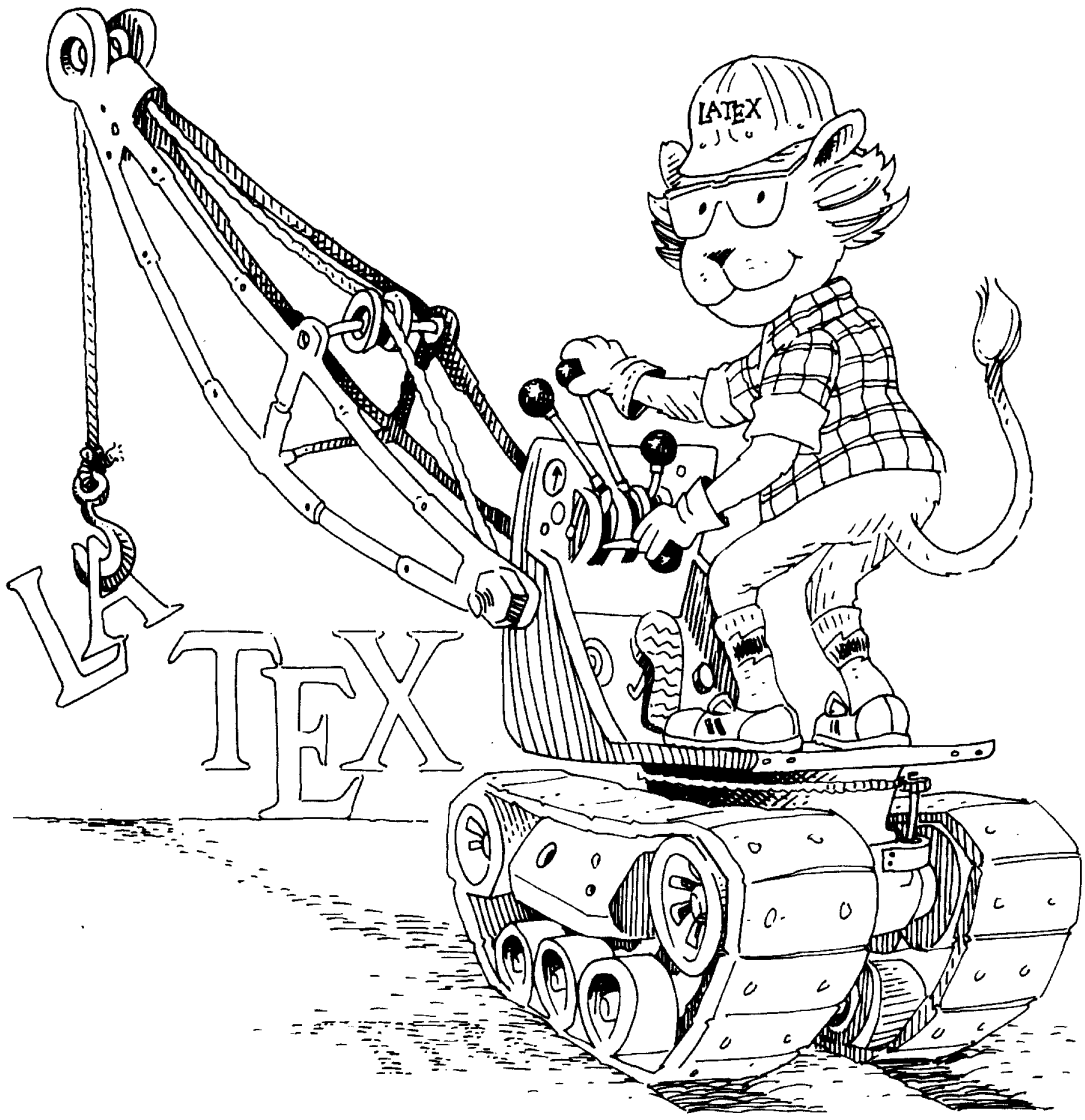
  For an up-to-date guide to LaTeX see the Wikibook on https://en.wikibooks.org/wiki/LaTeX

Attachments

- Latex_Lampor_chp1.pdf

CHAPTER 1

Getting Acquainted



LaTeX is a system for typesetting documents. Its first widely available version, mysteriously numbered 2.09, appeared in 1985. LaTeX is now extremely popular in the scientific and academic communities, and it is used extensively in industry. It has become a *lingua franca* of the scientific world; scientists send their papers electronically to colleagues around the world in the form of LaTeX input.

Over the years, various nonstandard enhancements were made to LaTeX 2.09 to overcome some of its limitations. LaTeX input that made use of these enhancements would not work properly at all sites. A new version of LaTeX was needed to keep a Tower of Babel from rising. The current version of LaTeX, with the somewhat less mysterious number 2_ε, was released in 1994. LaTeX 2_ε contains an improved method for handling different styles of type, commands for including graphics and producing colors, and many other new features.

Almost all standard LaTeX 2.09 input files will work with LaTeX 2_ε. However, to take advantage of the new features, users must learn a few new LaTeX 2_ε conventions. LaTeX 2.09 users should read Appendix D to find out what has changed. The rest of this book is about LaTeX, which, until a newer version appears, means LaTeX 2_ε.

LaTeX is available for just about any computer made today. The versions that run on these different systems are essentially the same; an input file created according to the directions in this book should produce the same output with any of them. However, how you actually run LaTeX depends upon the computer system. Moreover, some new features may not be available on all systems when LaTeX 2_ε is first released. For each computer system, there is a short companion to this book, titled something like *Local Guide to* LaTeX *for the McKludge PC*, containing information specific to that system. I will call it simply the *Local Guide*. It is distributed with the LaTeX software.

There is another companion to this book, *The* LaTeX *Companion* by Goossens, Mittelbach, and Samarin [3]. This companion is an in-depth guide to LaTeX and to its *packages*—standard enhancements that can be used at any site to provide additional features. The LaTeX *Companion* is the place to look if you can't find what you need in this book. It describes more than a hundred packages.

1.1 How to Avoid Reading This Book

Many people would rather learn about a program at their computer than by reading a book. There is a small sample LaTeX input file named `small2e.tex` that shows how to prepare your own input files for typesetting simple documents. Before reading any further, you might want to examine `small2e.tex` with a text editor and modify it to make an input file for a document of your own, then run LaTeX on this file and see what it produces. The *Local Guide* will tell you how to find `small2e.tex` and run LaTeX on your computer; it may also contain information about text editors. Be careful not to destroy the original version of `small2e.tex`; you'll probably want to look at it again.

The file `small2e.tex` is only forty lines long, and it shows how to produce only very simple documents. There is a longer file named `sample2e.tex` that contains more information. If `small2e.tex` doesn't tell you how to do something, you can try looking at `sample2e.tex`.

If you prefer to learn more about a program before you use it, read on. Almost everything in the sample input files is explained in the first two chapters of this book.

1.2 How to Read This Book

While `sample2e.tex` illustrates many of LaTeX's features, it is still only about two hundred lines long, and there is a lot that it doesn't explain. Eventually, you will want to typeset a document that requires more sophisticated formatting than you can obtain by imitating the two sample input files. You will then have to look in this book for the necessary information. You can read the section containing the information you need without having to read everything that precedes it. However, all the later chapters assume you have read Chapters 1 and 2. For example, suppose you want to set one paragraph of a document in small type. Looking up "type size" in the index or browsing through the table of contents will lead you to Section 6.7.1, which talks about "declarations" and their "scope"—simple concepts that are explained in Chapter 2. It will take just a minute or two to learn what to do if you've already read Chapter 2; it could be quite frustrating if you haven't. So, it's best to read the first two chapters now, before you need them.

LaTeX's input is a file containing the document's text together with commands that describe the document's structure; its output is a file of typesetting instructions. Another program must be run to convert these instructions into printed output. With a high-resolution printer, LaTeX can generate book-quality typesetting.

This book tells you how to prepare a LaTeX input file. The current chapter discusses the philosophy underlying LaTeX; here is a brief sketch of what's in the remaining chapters and appendices:

Chapter 2 explains what you should know to handle most simple documents and to read the rest of the book. Section 2.5 contains a summary of everything in the chapter; it serves as a short reference manual.

Chapter 3 describes logical structures for handling a variety of formatting problems. Section 3.4 explains how to define your own commands, which can save typing when you write the document and retyping when you change it. It's a good idea to read the introduction—up to the beginning of Section 3.1—before reading any other part of the chapter.

Chapter 4 contains features especially useful for large documents, including automatic cross-referencing and commands for splitting a large file into smaller pieces. Section 4.7 discusses sending your document electronically.

Chapter 5 is about making books, slides, and letters (the kind you send by post).

Chapter 6 describes the visual formatting of the text. It has information about changing the style of your document, explains how to correct bad line and page breaks, and tells how to do your own formatting of structures not explicitly handled by LaTeX.

Chapter 7 discusses pictures—drawing them yourself and inserting ones prepared with other programs—and color.

Chapter 8 explains how to deal with errors. This is where you should look when LaTeX prints an error message that you don't understand.

Appendix A describes how to use the *MakeIndex* program to make an index.

Appendix B describes how to make a bibliographic database for use with BibTeX, a separate program that provides an automatic bibliography feature for LaTeX.

Appendix C is a reference manual that compactly describes all LaTeX's features, including many advanced ones not described in the main text. If a command introduced in the earlier chapters seems to lack some necessary capabilities, check its description here to see if it has them. This appendix is a convenient place to refresh your memory of how something works.

Appendix D describes the differences between the current version of LaTeX and the original version, LaTeX 2.09.

Appendix E is for the reader who knows TeX, the program on which LaTeX is built, and wants to use TeX commands that are not described in this book.

When you face a formatting problem, the best place to look for a solution is in the table of contents. Browsing through it will give you a good idea of what LaTeX has to offer. If the table of contents doesn't work, look in the index; I have tried to make it friendly and informative.

Each section of Chapters 3–7 is reasonably self-contained, assuming only that you have read Chapter 2. Where additional knowledge is required, explicit cross-references are given. Appendix C is also self-contained, but a command's description may be hard to understand without first reading the corresponding description in the earlier chapters.

The descriptions of most LaTeX commands include examples of their use. In this book, examples are formatted in two columns, as follows:

The left column shows the printed output; the right column contains the input that produced it.

```
The left column shows the printed output;
the right column contains the input that
produced it.
```

Note the special typewriter type style in the right column. It indicates what you type—either text that you put in the input file or something like a file name that you type as part of a command to the computer.

Since the sample output is printed in a narrower column, and with smaller type, than LaTeX normally uses, it won't look exactly like the output you'd get from that input. The convention of the output appearing to the left of the corresponding input is generally also used when commands and their output are listed in tables.

1.3 The Game of the Name

The TeX in LaTeX refers to Donald Knuth's TeX typesetting system. The LaTeX program is a special version of TeX that understands LaTeX commands. Think of LaTeX as a house built with the lumber and nails provided by TeX. You don't need lumber and nails to live in a house, but they are handy for adding an extra room. Most LaTeX users never need to know any more about TeX than they can learn from this book. However, the lower-level TeX commands described in *The TeXbook* [4] can be very useful when creating a new package for LaTeX.

I will use the term "TeX" when describing standard TeX features and "LaTeX" when describing features unique to LaTeX, but the distinction will be of interest mainly to readers already familiar with TeX. You may ignore it and use the two names interchangeably.

One of the hardest things about using LaTeX is deciding how to pronounce it. This is also one of the few things I'm not going to tell you about LaTeX, since pronunciation is best determined by usage, not fiat. TeX is usually pronounced *teck*, making *lah*-teck, lah-*teck*, and *lay*-teck the logical choices; but language is not always logical, so *lay*-tecks is also possible.

The written word carries more legal complications than the spoken, and the need to distinguish TeX and LaTeX from similarly spelled products restricts how you may write them. The best way to refer to these programs is by their logos, which can be generated with simple LaTeX commands. When this is impossible, as in an e-mail message, you should write them as `TeX` and `LaTeX`, where the unusual capitalization identifies these computer programs.

1.4 Turning Typing into Typography

Traditionally, an author provides a publisher with a typed manuscript. The publisher's typographic designer decides how the manuscript is to be formatted, specifying the length of the printed line, what style of type to use, how much

space to leave above and below section headings, and many other things that determine the printed document's appearance. The designer writes a series of instructions to the typesetter, who uses them to decide where on the page to put each of the author's words and symbols. In the old days, typesetters produced a matrix of metal type for each page; today they produce computer files. In either case, their output is used to control the machine that does the actual typesetting.

LaTeX is your typographic designer, and TeX is its typesetter. The LaTeX commands that you type are translated into lower-level TeX typesetting commands. Being a modern typesetter, TeX produces a computer file, called the *device-independent* or `dvi` file. The *Local Guide* explains how to use this file to generate a printed document with your computer. It also explains how to view your document on your computer, using a screen previewer. Unless your document is very short, you will want to see the typeset version as you're writing it. Use a previewer instead of laying waste to our planet's dwindling forests by printing lots of intermediate versions. In fact, unless you want to take a copy with you on a wilderness expedition, you may never have to print it at all. It is easier and faster to distribute your document electronically than by mailing paper copies.

A human typographic designer knows what the manuscript is generally about and uses this knowledge in deciding how to format it. Consider the following typewritten manuscript:

```
The German mathematician Kronecker, sitting
quietly at his desk, wrote:
     God created the whole numbers; all
     the rest is man's work.
Seated in front of the terminal, with Basic
hanging on my every keystroke, I typed:
     for i = 1 to infinity
     let number[i] = i
```

A human designer knows that the first indented paragraph (`God created ...`) is a quotation and the second is a computer program, so the two should be formatted differently. He would probably set the quotation in ordinary roman type and the computer program in a typewriter type style. LaTeX is only a computer program and can't understand English, so it can't figure all this out by itself. It needs more help from you than a human designer would.

The function of typographic design is to help the reader understand the author's ideas. For a document to be easy to read, its visual structure must reflect its logical structure. Quotations and computer programs, being logically distinct structural elements, should be distinguished visually from one another. The designer should therefore understand the document's logical structure. Since LaTeX can't understand your prose, you must explicitly indicate the logical structure by typing special commands. The primary function of almost all the LaTeX

commands that you type should be to describe the logical structure of your document. As you are writing your document, you should be concerned with its logical structure, not its visual appearance. The LaTeX approach to typesetting can therefore be characterized as *logical design.*

1.5 Why LaTeX?

When LaTeX was introduced in 1985, few authors had the facilities for typesetting their own documents. Today, desktop publishing is commonplace. You can buy a "WYSIWYG" (what you see is what you get) program that lets you see exactly what your document will look like as you type it. WYSIWYG programs are very appealing. They make it easy to put text wherever you want in whatever size and style of type you want. Why use LaTeX, which requires you to tell it that a piece of text is a quotation or a computer program, when a WYSIWYG program allows you to format the text just the way you want it?

WYSIWYG programs replace LaTeX's logical design with *visual design.* Visual design is fine for short, simple documents like letters and memos. It is not good for more complex documents such as scientific papers. WYSIWYG has been characterized as "what you see is all you've got".[1] To illustrate the advantage of logical over visual design, I will consider a simple example from the file `sample2e.tex`.

Near the top of the second page of the document is the mathematical term (A, B). With a WYSIWYG program, this term is entered by typing `(A,B)`. You could type it the same way in the LaTeX input. However, the term represents a mathematical structure—the inner product of A and B. An experienced LaTeX user will define a command to express this structure. The file `sample2e.tex` defines the command `\ip` so that `\ip{A}{B}` produces (A, B). The term (Γ, ψ') near the end of the document is also an inner product and is produced with the `\ip` command.

Suppose you decide that there should be a little more space after the comma in an inner product. Just changing the definition of the `\ip` command will change (A,B) to $(A,\ B)$ and (Γ,ψ') to $(\Gamma,\ \psi')$. With a WYSIWYG program, you would have to insert the space by hand in each formula—not a problem for a short document with two such terms, but a mathematical paper could contain dozens and a book could contain hundreds. You would probably produce inconsistent formatting by missing some formulas or forgetting to add the space when entering new ones. With LaTeX, you don't have to worry about formatting while writing your document. Formatting decisions can be made and changed at any time.

The advantage of logical design becomes even more obvious if you decide that you prefer the notation $\langle A|B\rangle$ for the inner product of A and B. The

[1] Brian Reid attributes this phrase to himself and/or Brian Kernighan.

file `sample2e.tex` contains an alternate definition of `\ip` that produces this notation.

Typing `\ip{A}{B}` is just a little more work than typing `(A,B)` (though it's a lot easier than entering $\langle A|B\rangle$ if the symbols "⟨" and "⟩" must be chosen with a mouse from a pull-down menu). But this small effort is rewarded by the benefits of maintaining the logical structure of your document instead of just its visual appearance.

One advantage of WYSIWYG programs is that you can see the formatted version of your document while writing it. Writing requires reading what you have already written. Although you want LaTeX to know that the term is an inner product, you would like to read (A, B) or $\langle A|B\rangle$, not `\ip{A}{B}`. The speed of modern computers has eliminated much of this advantage. I now type a couple of keystrokes and, a few seconds later, a typeset version of the section I am working on appears on my screen. As computers get faster, those few seconds will turn into a fraction of a second.

1.6 Turning Ideas into Input

The purpose of writing is to present your ideas to the reader. This should always be your primary concern. It is easy to become so engrossed with form that you neglect content. Formatting is no substitute for writing. Good ideas couched in good prose will be read and understood, regardless of how badly the document is formatted. LaTeX was designed to free you from formatting concerns, allowing you to concentrate on writing. If you spend a lot of time worrying about form, you are misusing LaTeX.

Even if your ideas are good, you can probably learn to express them better. The classic introduction to writing English prose is Strunk and White's brief *Elements of Style* [6]. A more complete guide to using language properly is Theodore Bernstein's *The Careful Writer* [1]. These two books discuss general writing style. Writers of scholarly or technical prose need additional information. Mary-Claire van Leunen's *Handbook for Scholars* [7] is a delightful guide to academic and scholarly writing. The booklet titled *How to Write Mathematics* [5] can help scientists and engineers as well as mathematicians. It's also useful to have a weightier reference book at hand; *Words into Type* [8] and the *Chicago Manual of Style* [2] are two good ones.

1.7 Trying It Out

You may already have run LaTeX with input based on the sample files. If not, this is a good time to learn how. The section in the *Local Guide* titled *Running a Sample File* explains how to obtain a copy of the file `sample2e.tex` and run LaTeX with it as input. Follow the directions and see what LaTeX can do.

After printing the document generated in this way, try changing the document's format. Using a text editor, examine the file `sample2e.tex`. A few lines down from the beginning of the file is a line that reads

```
\documentclass{article}
```

Change that line to

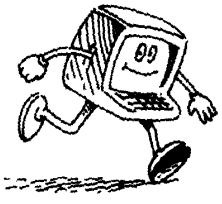
```
\documentclass[twocolumn]{article}
```

Save the changed file under the name `chgsam.tex`, and use this file to print a new version of the document. To generate the new version, do exactly what you did the last time, except type `chgsam` wherever you had typed `sample2e`. Comparing the two printed versions shows how radically the appearance of the document can be altered by a simple change to a command. To try still another format, change `chgsam.tex` so the line above reads

```
\documentclass[12pt]{article}
```

and use the changed file to print a third version of the document.

From now on, I will usually ignore the process of going from the LaTeX input file to the printed output and will write something like: "Typing --- produces a long dash." What this really means is that putting the three characters --- in your input file will, when that file is processed by LaTeX and the device-independent file printed, produce a long dash in the printed output.



Conferences floss art school - Media Design

Type	Web Page
URL	http://pzwiki.wdka.nl/mediadesign/Conferences_floss_art_school
Accessed	10/25/2016, 6:19:33 PM
Date Added	10/25/2016, 6:19:33 PM
Modified	10/25/2016, 6:20:01 PM

Notes:

- Floss Art School

  â€œConferences Floss Art School - Media Design.â€ Accessed October 25, 2016. http://pzwiki.wdka.nl/mediadesign/Conferences_floss_art_school.

```
wget --quiet --output-document=FLOSS_Art-school.html  "http://pzwiki.wdka.nl/mw-mediadesign/index.php?title=Conferences_floss_art_school&action=render"; pandoc FLOSS_Art-school.html -
```

Attachments

- FLOSS_Art-school.pdf

Contents

How to run an art school on Free Software/Open Source?

(Florian Cramer, Aymeric Mansoux, Michael Murtaugh, 2010)

Introduction

We each introduce ourselves briefly.

Florian

- Images: website, students in classroom, gordo in studio

We have to admit that this title is a hyperbole. Actually, we are not talking about our entire art school, but only one study programme: the Networked Media Master at the Piet Zwart Institute of the Willem de Kooning Academy in Rotterdam, the Netherlands. It is a small, very international and cross-disciplinary new media design and art programme in the graduate school of Rotterdam's traditional art academy. Our students have backgrounds as graphic and web designers, media artists and activists, but also include architects, fine artists and even a dancer. Our common interest is to critically think about digital and computer media, and create one's own media work based on that thinking and research. The most simple formula we use is the following: it's media design as design of media, not just with media. And that's where Free Software and Open Source come in - because they provide the building blocks for these self-created media.

- Light writer: Ricardo Lafuente: a self-created medium on the basis of the Arduino board
- Web 2.0 Suicide Machine: Gordon Savicic/Danja Vasiliev: a Web application that thoroughly deletes your social media profiles.

Aymeric

So when we talk about Free and Open Source (FLOSS) in our department, we do not simply mean the scenario of replacing Photoshop with The Gimp, Max/MSP with Pure Data, Cinema 4d with Blender, and so on. We are more interested in FLOSS as an entry point into a different media practice - based on a comprehensive critical rethinking of communication in its relation to technology. Apart from that, we have a very practical interest in the non-mainstream tools and work flows provided by Open Source and Free Software (ref web 2.0 suicide machine). This puts us into a different camp than even the GNU Project of the Free Software Foundation because our concern is not to obtain free alternatives to existing software, no matter how this software is designed.

Media Design?

Michael

- Ivan Monroy Lopez, man immigration

(Former co-course director) Matthew Fuller would use the example of "vapourware", software that is announced and discussed without ever being actually released to the public (or perhaps even being written at all), as a simple example of how software is a cultural, and not just technical phenomenon.

Florian

- Darija Medic: The "us" in virus, artistic/activist reflection of openness in the medium of a "viral" sticker campaign

For example this piece evokes software, while taking the form of stickers.

Michael

Using open source in art education is also about using new methodologies and approaches to teaching. Traditionally "good design" is equated with notions of "simplicity" and "seamlessness". The Faux-metal skins popular in current interface design seems to respond to a need to reassure the user of a stability and sturdiness. Free software often reveals the underlying assumptions and decisions that have been in the design of software and confronts the user take an active position in how they want to work, with what tools on what terms. Open source confronts students with the fact that software is developed in communities, with differing philosophies / approaches / priorities.

- Epicpedia: Annemieke van der Hoek

Presents wikipedia articles as a dramatic conversation over time, as opposed to a seamless essay.

For me personally, as a software developer, the experience of working with free software has encouraged me to shift from a "make everything from scratch" way of thinking to one that is more modular, and which looks towards tapping into existing code and software communities; I've come to value the creative potential of simply making novel connections between existing systems; In addition, conceiving a project as a pipeline opens it up to collaboration, and broadens the range of your work beyond your own particular skills or interests.

Aymeric

The pipeline, as an approach to creating an artistic work, is in dramatic contrast to a traditional image of the "isolated artist working in the 'clean room' of his/her creative suite. . . " and recasts work as being a flow of material across different sources. aknowledges that the tools are themselves represent decisions, assumptions, work of others, negotations/compromises

Often a project can have a powerful impact simply by making an unexpected connection between systems.

- Timo Klok (in collaboration): Pirates of the Amazon

(many ppl thought it was a crack.)

Florian

We are also critical about the typical divide between designers on the one side and engineers on the other. It's the classical new media trap where the artist develops the vision, the technician the code, and the result is disappointing because neither speak each other's language.

Networked Media Design

Florian

You could say that FLOSS Artistic tools are not as professional as the Adobe suite. I think we are in the same situation as database servers in the 1990's. MySQL was criticized for lacking the full feature set of Oracle, just as the GIMP is now criticized for lacking the features of Photoshop. The breakthough of MySQL came with the rise of the web, and the need to have simple flexible solutions without licensing fees. Unlike photoshop, the gimp can be run on a server to generate graphics in real time. In addition, the growing important generative design creates an ideal situation for Open source tools.

Aymeric

One of the key aspect of the program is to get students to question their work flows, their tools, their assumptions. A key challenge is getting students to stop thinking in terms of "what can the software do for me", and switch the mentality to "what can I do with software",and eventually create their own programs. This means to pull students out of a production mode of "getting things done" and into a more reflective manner of work. The complexity inherent to using Free Software is often very good for this purpose. (Even if we are repeating ourselves here, it's still important to keep stressing it:) We would like to encourage FLOSS developers not to strive for a better Photoshop, a better Illustrator or a better Final Cut Pro, but build artistic design tools on the traditional virtues of programmable and networked Free Software. We need more software projects that are low level enough to allow artists and designers to develop their your own GUI metaphors, command line tools and of course artistic software, while - at the same time - being accessible and usable without a degree in Computer Science. Software like the Unix/GNU text tools, ImageMagick, and frameworks like MLT really shine in this respect.

Florian

These are concrete practical issues for us. But there's also the level of media theory and criticism which is integral to our study programme and the way we

work and think. Free Software and Open Source is useful in this context, too. It can serve as a critical tool because it cuts into all major social, economic, political and artistic issues of information ownership, media governance and participation. However, it is no magical bullet.

Looking at the founding manifesto of the Open Source movement, Eric S. Raymond's "The Cathedral and the Bazaar", we see that it is based on the notion that an open system, or a free flow of information and labor, will result in a self-regulating whole providing optimal solutions for everyone. Today, we now that this has been over-optimistic thinking of the 1990s.

Aymeric

If we see where Free Software and Open Source are today, more than ten years after Raymond's manifesto, then some questions need to be asked: In the Internet 'cloud', in all kinds of embedded devices from routers to media players, and now on mobile phones, Free Software is mostly used as a cheap productivity stack underneath proprietary technology. The "world domination" it achieved this way is quite different from the one imagined in the 1990s. But investigating such questions is exactly what makes a study programme like ours more engaging, and hopefully helps us to have a larger vision of media and design.

Under the Hood

Michael

(image: mac_issues)

Early in the course, we have sessions to install Linux onto student's laptops (often leaving their original OS – Mac or Windows – intact). The experience is an important one – a key moment to confront students with a question of what exactly is the computer in front of them?

(image: Ted Nelson: Computer Lib)

Breaking through the glossy veneer of a polished operating system designed to "just work", is a crucial first step in understanding the computer and it's software as a socially constructed assemblage: of electronic components, of software, of legal agreements all with a particular history.

(image: Danja: meme 2.0) (for example this is Danja Vasilijev's implementation of a web server as a physical object)

Ultimately it's about instilling a sense of empowerment as what was previously a "magic box", something you ought not to tamper with, becomes a platform for actively re-imagining / rethinking what computers and software can be.

Aymeric

There, were, however a number of issues:

- Since we used Gentoo as our standard distribution, Gentoo quirks and were falsely perceived as Linux and Open Source quirks.
- Laptops and their complete driver support through the Linux kernel were a problem, and still are a problem.
- The tech barriers and learning curves are high, particularly for students trained in graphic design Bachelor programs.
- Students who professionally work as graphic designers will still need their proprietary tools. A graphic designer will continue to work with InDesign for non-generative design of printed matters for this reason.
- Issue Magazine: Alexandre Leray, Stephanie Vilayphiou; interview with David Reinfurt addressed the problems of open source graphic design)
- Peer to peer design strategies: Emanuele Bonetti

These issues are less pronounced for web-based work. Linux and FLOSS are the software that drives the Internet. If students develop web applications, then this is the technology they need to learn as media designers and artists.

Florian

We fully switched to GNU/Linux when we decided that in a media study program computers are instruments much like musical instruments in a conservatory. Just as every music student brings their own instrument, we asked every student to bring their own laptop to the course, and provided Linux installation support. In that year, Linux broke through as operating system used by our students - and not just by staff - because it ran on all machines no matter whether originally designed for Mac OS or Windows. As a lingua franca, it allowed our students to better exchange knowledge and help each other. We also got a whole generation of students who appreciated Linux for actually being different, instead of just claiming to “think different”.

Aymeric

Some details: (images)

- Wiki-based code cookbook - made from in-house knowledge and inspired recipes from sources such as ThinkPython.
- Wiki-based planning of the course content and direction - from thematic project to the writing of this paper.

- Wiki-based sandbox for the students - essay drafts, notes during tutorials, assignments.
- Distributed version control for the code developed by the second years - we use Git.
- Dual boot or single boot to GNU/Linux for students - we use Ubuntu for its practical advantages over less desktop-friendly distros, but actively encourage our students to break it apart, remove bloat and customize their system later on.
- 2 Debian servers with SSH accounts for all students - servers are used as networked sandboxes and production hosts for their code.
- university-hosted web site/blog for news on the study programme, and self-hosted blogs for student research projects.
- Free software licensing - we recommend to use GPLv3 and AGPLv3 for their projects, next the common set of free culture licenses available, but we obviously let the students choose for themselves.

Concluding Example: Shahee Ilyas, Framing Leaders

- Framing Leaders: Shahee Ilyas

Shahee scraped data from Wikipedia pages, an visualized the length of time leaders have been in office by the width of their frame (the longer in power, the larger the frame). At the time of the final exhibition, Maumoon Gayoom, the leader of the Maldives was in the third position (having been in power since 1978). In 2008, Gayoom lost the presidential election.

- Wikipedia edit, Shehee Ilyas

Also: [1], as an example of how a simple assignment (make an edit on Wikipedia) lead Shahee to upload a personal picture (which he took from the cockpit of a plane once when returning home) on to the page of the Maldives; I believe the positive reponse to the picture (to judge by its history of use in Wikipedia) helped contribute to Shahee's interest in working with Wikipedia data in his final project.

- Early course assignment to make an edit on Wikipedia.
- Shahee, chose to upload an image he shot from the cockpit of an airplane when he was returning home to the Maldives.
- He posted the image using a Creative Common license.
- Through the wikipedia interface one can trace the history – including its being selected as a "featured image" by several different wikipedia language communities.

Translating FLOSS into the School Context (Conclusions)

Florian

A school is not an open-source project, nevertheless good lessons can be learned from open-source development:

Michael

- Pipeline: the power of novel connections
- Design isn't about slickness and seamlessness, but about systems

Aymeric

- 'Release early, release often': communicate, release, document and archive what you do using free culture licenses
- Don't mystify creation
- Do not design from scratch, but reuse work - 'Dwarfs standing on the shoulders of giants'
- Week project collaboration outside the institution

Florian

These are not only good principles for advanced art school education, but also very healthy recipes for the art world in general.

Bibliography

- Lawrence Liang, A Guide to Open Content Licenses, Piet Zwart Institute and de Waag, Rotterdam and Amsterdam 2004
- Freestyle - FLOSS in Design, workshop by the Piet Zwart Institute, 2004, transcripts on http://pzwart.wdka.hro.nl/mdr/Seminars2/freestyletrans1/view + http://pzwart.wdka.hro.nl/mdr/Seminars2/freestyletrans2/view
- FLOSS+Art. Ed. Aymeric Mansoux, Marloes de Valk, Openmute, London 2008 (PDF: http://people.makeart.goto10.org)
- Tools to fight boredom. Marloes de Valk, in Volume 28, Issue 1, 2009 of the Contemporary Music Review. Ed. Nick Collins and Andrew R. Brown, Routlegdge, London 2009 (pre-typeset version: http://pi.kuri.mu/tools-to-fight-boredom)
- Rock, Paper, Scissors and Floppy Disks. Anne Laforet, Aymeric Mansoux, Marloes de Valk, in Archive 2020. Sustainable archiving of born digital cultural content. Ed. Annet Dekker, Virtueel Platform, Amsterdam 2010 (pre-typeset version: http://pi.kuri.mu/rock)

- Florian Cramer, Free Software as Collaborative Text (2000), in: Sarai Reader 01, The Public Domain, New Delhi/Amsterdam 2001, p. 199-206

- # The GNU Manifesto

Type	Web Page
Author	Stallman Richard
URL	https://www.gnu.org/gnu/manifesto.html
Accessed	10/26/2016, 8:19:24 AM
Date Added	10/26/2016, 8:19:24 AM
Modified	10/26/2016, 8:31:40 AM

Attachments

- Snapshot

The GNU Manifesto

The GNU Manifesto (which appears below) was written by Richard Stallman in 1985 to ask for support in developing the GNU operating system. Part of the text was taken from the original announcement of 1983. Through 1987, it was updated in minor ways to account for developments; since then, it seems best to leave it unchanged.

Since that time, we have learned about certain common misunderstandings that different wording could help avoid. Footnotes added since 1993 help clarify these points.

If you want to install the GNU/Linux system, we recommend you use one of the 100% free software GNU/Linux distributions. For how to contribute, see http://www.gnu.org/help.

The GNU Project is part of the Free Software Movement, a campaign for freedom for users of software. It is a mistake to associate GNU with the term “open source”—that term was coined in 1998 by people who disagree with the Free Software Movement's ethical values. They use it to promote an amoral approach to the same field.

What's GNU? Gnu's Not Unix!

GNU, which stands for Gnu's Not Unix, is the name for the complete Unix-compatible software system which I am writing so that I can give it away free to everyone who can use it.(1) Several other volunteers are helping me. Contributions of time, money, programs and equipment are greatly needed.

So far we have an Emacs text editor with Lisp for writing editor commands, a source level debugger, a yacc-compatible parser generator, a linker, and around 35 utilities. A shell (command interpreter) is nearly completed. A new portable optimizing C compiler has compiled itself and may be released this year. An initial kernel exists but many more features are needed to emulate Unix. When the kernel and compiler are finished, it will be possible to distribute a GNU system suitable for program development. We will use TeX as our text formatter, but an nroff is being worked on. We will use the free, portable X Window System as well. After this we will add a portable Common Lisp, an Empire game, a spreadsheet, and hundreds of other things, plus online documentation. We hope to supply, eventually, everything useful that normally comes with a Unix system, and more.

GNU will be able to run Unix programs, but will not be identical to Unix. We will make all improvements that are convenient, based on our experience with other operating systems. In particular, we plan to have longer file names, file version numbers, a crashproof file system, file name completion perhaps, terminal-independent display support, and perhaps eventually a Lisp-based window system through which several Lisp programs and ordinary Unix programs can share a screen. Both C and Lisp will be available as system programming languages. We will try to support UUCP, MIT Chaosnet, and Internet protocols for communication.

GNU is aimed initially at machines in the 68000/16000 class with virtual memory, because they are the easiest machines to make it run on. The extra effort to make it run on smaller machines will be left to someone who wants to use it on them.

To avoid horrible confusion, please pronounce the *g* in the word “GNU” when it is the name of this project.

Why I Must Write GNU

I consider that the Golden Rule requires that if I like a program I must share it with other people who like it. Software sellers want to divide the users and conquer them, making each user agree not to share with others. I refuse to break solidarity with other users in this way. I cannot in good conscience sign a nondisclosure agreement or a software license agreement. For years I worked within the Artificial Intelligence Lab to resist such tendencies and other inhospitalities, but eventually they had gone too far: I could not remain in an institution where such things are done for me against my will.

So that I can continue to use computers without dishonor, I have decided to put together a sufficient body of free software so that I will be able to get along without any software that is not free. I have resigned from the AI Lab to deny MIT any legal excuse to prevent me from giving GNU away.(2)

Why GNU Will Be Compatible with Unix

Unix is not my ideal system, but it is not too bad. The essential features of Unix seem to be good ones, and I think I can fill in what Unix lacks without spoiling them. And a system compatible with Unix would be convenient for many other people to adopt.

How GNU Will Be Available

GNU is not in the public domain. Everyone will be permitted to modify and redistribute GNU, but no distributor will be allowed to restrict its further redistribution. That is to say, proprietary modifications will not be allowed. I want to make sure that all versions of GNU remain free.

Why Many Other Programmers Want to Help

I have found many other programmers who are excited about GNU and want to help.

Many programmers are unhappy about the commercialization of system software. It may enable them to make more money, but it requires them to feel in conflict with other programmers in general rather than feel as comrades. The fundamental act of friendship among programmers is the sharing of programs; marketing arrangements now typically used essentially forbid programmers to treat others as friends. The purchaser of software must choose between friendship and obeying the law. Naturally, many decide that friendship is more important. But those who believe in law often do not feel at ease with either choice. They become cynical and think that programming is just a way of making money.

By working on and using GNU rather than proprietary programs, we can be hospitable to everyone and obey the law. In addition, GNU serves as an example to inspire and a banner to rally others to join us in sharing. This can give us a feeling of harmony which is impossible if we use software that is not free. For about half the programmers I talk to, this is an important happiness that money cannot replace.

How You Can Contribute

> (Nowadays, for software tasks to work on, see the High Priority Projects list and the GNU Help Wanted list, the general task list for GNU software packages. For other ways to help, see the guide to helping the GNU operating system.)

I am asking computer manufacturers for donations of machines and money. I'm asking individuals for donations of programs and work.

One consequence you can expect if you donate machines is that GNU will run on them at an early date. The machines should be complete, ready to use systems, approved for use in a residential area, and not in need of sophisticated cooling or power.

I have found very many programmers eager to contribute part-time work for GNU. For most projects, such part-time distributed work would be very hard to coordinate; the independently written parts would not work together. But for the particular task of replacing Unix, this problem is absent. A complete Unix system contains hundreds of utility programs, each of which is documented separately. Most interface specifications are fixed by Unix compatibility. If each contributor can write a compatible replacement for a single Unix utility, and make it work properly in place of the original on a Unix system, then these utilities will work right when put together. Even allowing for Murphy to create a few unexpected problems, assembling these components will be a feasible task. (The kernel will require closer communication and will be worked on by a small, tight group.)

If I get donations of money, I may be able to hire a few people full or part time. The salary won't be high by programmers' standards, but I'm looking for people for whom building community spirit is as important as making money. I view this as a way of enabling dedicated people to devote their full energies to working on GNU by sparing them the need to make a living in another way.

Why All Computer Users Will Benefit

Once GNU is written, everyone will be able to obtain good system software free, just like air.(3)

This means much more than just saving everyone the price of a Unix license. It means that much wasteful duplication of system programming effort will be avoided. This effort can go instead into advancing the state of the art.

Complete system sources will be available to everyone. As a result, a user who needs changes in the system will always be free to make them himself, or hire any available programmer or company to make them for him. Users will no longer be at the mercy of one programmer or company which owns the sources and is in sole position to make changes.

Schools will be able to provide a much more educational environment by encouraging all students to study and improve the system code. Harvard's computer lab used to have the policy that no program could be installed on the system if its sources were not on public display, and upheld it by actually refusing to install certain programs. I was very much inspired by this.

Finally, the overhead of considering who owns the system software and what one is or is not entitled to do with it will be lifted.

Arrangements to make people pay for using a program, including licensing of copies, always incur a tremendous cost to society through the cumbersome mechanisms necessary to figure out how much (that is, which programs) a person must pay for. And only a police state can force everyone to obey them. Consider a space station where air must be manufactured at great cost: charging each breather per liter of air may be fair, but wearing the metered gas mask all day and all night is intolerable even if everyone can afford to pay the air bill. And the TV cameras everywhere to see if you ever take the mask off are outrageous. It's better to support the air plant with a head tax and chuck the masks.

Copying all or parts of a program is as natural to a programmer as breathing, and as productive. It ought to be

as free.

Some Easily Rebutted Objections to GNU's Goals

“Nobody will use it if it is free, because that means they can't rely on any support.”

“You have to charge for the program to pay for providing the support.”

If people would rather pay for GNU plus service than get GNU free without service, a company to provide just service to people who have obtained GNU free ought to be profitable.(4)

We must distinguish between support in the form of real programming work and mere handholding. The former is something one cannot rely on from a software vendor. If your problem is not shared by enough people, the vendor will tell you to get lost.

If your business needs to be able to rely on support, the only way is to have all the necessary sources and tools. Then you can hire any available person to fix your problem; you are not at the mercy of any individual. With Unix, the price of sources puts this out of consideration for most businesses. With GNU this will be easy. It is still possible for there to be no available competent person, but this problem cannot be blamed on distribution arrangements. GNU does not eliminate all the world's problems, only some of them.

Meanwhile, the users who know nothing about computers need handholding: doing things for them which they could easily do themselves but don't know how.

Such services could be provided by companies that sell just handholding and repair service. If it is true that users would rather spend money and get a product with service, they will also be willing to buy the service having got the product free. The service companies will compete in quality and price; users will not be tied to any particular one. Meanwhile, those of us who don't need the service should be able to use the program without paying for the service.

“You cannot reach many people without advertising, and you must charge for the program to support that.”

“It's no use advertising a program people can get free.”

There are various forms of free or very cheap publicity that can be used to inform numbers of computer users about something like GNU. But it may be true that one can reach more microcomputer users with advertising. If this is really so, a business which advertises the service of copying and mailing GNU for a fee ought to be successful enough to pay for its advertising and more. This way, only the users who benefit from the advertising pay for it.

On the other hand, if many people get GNU from their friends, and such companies don't succeed, this will show that advertising was not really necessary to spread GNU. Why is it that free market advocates don't want to let the free market decide this?(5)

“My company needs a proprietary operating system to get a competitive edge.”

GNU will remove operating system software from the realm of competition. You will not be able to get an edge in this area, but neither will your competitors be able to get an edge over you. You and they will compete in other areas, while benefiting mutually in this one. If your business is selling an operating system, you will not like GNU, but that's tough on you. If your business is something else, GNU can save you from being pushed into the expensive business of selling operating systems.

I would like to see GNU development supported by gifts from many manufacturers and users, reducing the cost to each.(6)

“Don't programmers deserve a reward for their creativity?”

If anything deserves a reward, it is social contribution. Creativity can be a social contribution, but only in so far as society is free to use the results. If programmers deserve to be rewarded for creating innovative programs, by the same token they deserve to be punished if they restrict the use of these programs.

“Shouldn't a programmer be able to ask for a reward for his creativity?”

There is nothing wrong with wanting pay for work, or seeking to maximize one's income, as long as one does not use means that are destructive. But the means customary in the field of software today are based on destruction.

Extracting money from users of a program by restricting their use of it is destructive because the restrictions reduce the amount and the ways that the program can be used. This reduces the amount of wealth that humanity derives from the program. When there is a deliberate choice to restrict, the harmful consequences are deliberate destruction.

The reason a good citizen does not use such destructive means to become wealthier is that, if everyone did so, we would all become poorer from the mutual destructiveness. This is Kantian ethics; or, the Golden Rule. Since I

do not like the consequences that result if everyone hoards information, I am required to consider it wrong for one to do so. Specifically, the desire to be rewarded for one's creativity does not justify depriving the world in general of all or part of that creativity.

“Won't programmers starve?”

I could answer that nobody is forced to be a programmer. Most of us cannot manage to get any money for standing on the street and making faces. But we are not, as a result, condemned to spend our lives standing on the street making faces, and starving. We do something else.

But that is the wrong answer because it accepts the questioner's implicit assumption: that without ownership of software, programmers cannot possibly be paid a cent. Supposedly it is all or nothing.

The real reason programmers will not starve is that it will still be possible for them to get paid for programming; just not paid as much as now.

Restricting copying is not the only basis for business in software. It is the most common basis(7) because it brings in the most money. If it were prohibited, or rejected by the customer, software business would move to other bases of organization which are now used less often. There are always numerous ways to organize any kind of business.

Probably programming will not be as lucrative on the new basis as it is now. But that is not an argument against the change. It is not considered an injustice that sales clerks make the salaries that they now do. If programmers made the same, that would not be an injustice either. (In practice they would still make considerably more than that.)

“Don't people have a right to control how their creativity is used?”

“Control over the use of one's ideas” really constitutes control over other people's lives; and it is usually used to make their lives more difficult.

People who have studied the issue of intellectual property rights(8) carefully (such as lawyers) say that there is no intrinsic right to intellectual property. The kinds of supposed intellectual property rights that the government recognizes were created by specific acts of legislation for specific purposes.

For example, the patent system was established to encourage inventors to disclose the details of their inventions. Its purpose was to help society rather than to help inventors. At the time, the life span of 17 years for a patent was short compared with the rate of advance of the state of the art. Since patents are an issue only among manufacturers, for whom the cost and effort of a license agreement are small compared with setting up production, the patents often do not do much harm. They do not obstruct most individuals who use patented products.

The idea of copyright did not exist in ancient times, when authors frequently copied other authors at length in works of nonfiction. This practice was useful, and is the only way many authors' works have survived even in part. The copyright system was created expressly for the purpose of encouraging authorship. In the domain for which it was invented—books, which could be copied economically only on a printing press—it did little harm, and did not obstruct most of the individuals who read the books.

All intellectual property rights are just licenses granted by society because it was thought, rightly or wrongly, that society as a whole would benefit by granting them. But in any particular situation, we have to ask: are we really better off granting such license? What kind of act are we licensing a person to do?

The case of programs today is very different from that of books a hundred years ago. The fact that the easiest way to copy a program is from one neighbor to another, the fact that a program has both source code and object code which are distinct, and the fact that a program is used rather than read and enjoyed, combine to create a situation in which a person who enforces a copyright is harming society as a whole both materially and spiritually; in which a person should not do so regardless of whether the law enables him to.

“Competition makes things get done better.”

The paradigm of competition is a race: by rewarding the winner, we encourage everyone to run faster. When capitalism really works this way, it does a good job; but its defenders are wrong in assuming it always works this way. If the runners forget why the reward is offered and become intent on winning, no matter how, they may find other strategies—such as, attacking other runners. If the runners get into a fist fight, they will all finish late.

Proprietary and secret software is the moral equivalent of runners in a fist fight. Sad to say, the only referee we've got does not seem to object to fights; he just regulates them (“For every ten yards you run, you can fire one shot”). He really ought to break them up, and penalize runners for even trying to fight.

“Won't everyone stop programming without a monetary incentive?”

Actually, many people will program with absolutely no monetary incentive. Programming has an irresistible fascination for some people, usually the people who are best at it. There is no shortage of professional musicians who keep at it even though they have no hope of making a living that way.

But really this question, though commonly asked, is not appropriate to the situation. Pay for programmers will not disappear, only become less. So the right question is, will anyone program with a reduced monetary incentive? My experience shows that they will.

For more than ten years, many of the world's best programmers worked at the Artificial Intelligence Lab for far less money than they could have had anywhere else. They got many kinds of nonmonetary rewards: fame and appreciation, for example. And creativity is also fun, a reward in itself.

Then most of them left when offered a chance to do the same interesting work for a lot of money.

What the facts show is that people will program for reasons other than riches; but if given a chance to make a lot of money as well, they will come to expect and demand it. Low-paying organizations do poorly in competition with high-paying ones, but they do not have to do badly if the high-paying ones are banned.

“We need the programmers desperately. If they demand that we stop helping our neighbors, we have to obey.”

You're never so desperate that you have to obey this sort of demand. Remember: millions for defense, but not a cent for tribute!

“Programmers need to make a living somehow.”

In the short run, this is true. However, there are plenty of ways that programmers could make a living without selling the right to use a program. This way is customary now because it brings programmers and businessmen the most money, not because it is the only way to make a living. It is easy to find other ways if you want to find them. Here are a number of examples.

A manufacturer introducing a new computer will pay for the porting of operating systems onto the new hardware.

The sale of teaching, handholding and maintenance services could also employ programmers.

People with new ideas could distribute programs as freeware(9), asking for donations from satisfied users, or selling handholding services. I have met people who are already working this way successfully.

Users with related needs can form users' groups, and pay dues. A group would contract with programming companies to write programs that the group's members would like to use.

All sorts of development can be funded with a Software Tax:

Suppose everyone who buys a computer has to pay x percent of the price as a software tax. The government gives this to an agency like the NSF to spend on software development.

But if the computer buyer makes a donation to software development himself, he can take a credit against the tax. He can donate to the project of his own choosing—often, chosen because he hopes to use the results when it is done. He can take a credit for any amount of donation up to the total tax he had to pay.

The total tax rate could be decided by a vote of the payers of the tax, weighted according to the amount they will be taxed on.

The consequences:

- The computer-using community supports software development.
- This community decides what level of support is needed.
- Users who care which projects their share is spent on can choose this for themselves.

In the long run, making programs free is a step toward the postscarcity world, where nobody will have to work very hard just to make a living. People will be free to devote themselves to activities that are fun, such as programming, after spending the necessary ten hours a week on required tasks such as legislation, family counseling, robot repair and asteroid prospecting. There will be no need to be able to make a living from programming.

We have already greatly reduced the amount of work that the whole society must do for its actual productivity, but only a little of this has translated itself into leisure for workers because much nonproductive activity is required to accompany productive activity. The main causes of this are bureaucracy and isometric struggles against competition. Free software will greatly reduce these drains in the area of software production. We must do this, in order for technical gains in productivity to translate into less work for us.

Footnotes

1. The wording here was careless. The intention was that nobody would have to pay for **permission** to use the GNU system. But the words don't make this clear, and people often interpret them as saying that copies of GNU should always be distributed at little or no charge. That was never the intent; later on, the manifesto mentions the possibility of companies providing the service of distribution for a profit. Subsequently I have learned to distinguish carefully between “free” in the sense of freedom and “free” in

the sense of price. Free software is software that users have the freedom to distribute and change. Some users may obtain copies at no charge, while others pay to obtain copies—and if the funds help support improving the software, so much the better. The important thing is that everyone who has a copy has the freedom to cooperate with others in using it.

2. The expression “give away” is another indication that I had not yet clearly separated the issue of price from that of freedom. We now recommend avoiding this expression when talking about free software. See “Confusing Words and Phrases” for more explanation.
3. This is another place I failed to distinguish carefully between the two different meanings of “free”. The statement as it stands is not false—you can get copies of GNU software at no charge, from your friends or over the net. But it does suggest the wrong idea.
4. Several such companies now exist.
5. Although it is a charity rather than a company, the Free Software Foundation for 10 years raised most of its funds from its distribution service. You can order things from the FSF to support its work.
6. A group of computer companies pooled funds around 1991 to support maintenance of the GNU C Compiler.
7. I think I was mistaken in saying that proprietary software was the most common basis for making money in software. It seems that actually the most common business model was and is development of custom software. That does not offer the possibility of collecting rents, so the business has to keep doing real work in order to keep getting income. The custom software business would continue to exist, more or less unchanged, in a free software world. Therefore, I no longer expect that most paid programmers would earn less in a free software world.
8. In the 1980s I had not yet realized how confusing it was to speak of “the issue” of “intellectual property”. That term is obviously biased; more subtle is the fact that it lumps together various disparate laws which raise very different issues. Nowadays I urge people to reject the term “intellectual property” entirely, lest it lead others to suppose that those laws form one coherent issue. The way to be clear is to discuss patents, copyrights, and trademarks separately. See further explanation of how this term spreads confusion and bias.
9. Subsequently we learned to distinguish between “free software” and “freeware”. The term “freeware” means software you are free to redistribute, but usually you are not free to study and change the source code, so most of it is not free software. See “Confusing Words and Phrases” for more explanation.