PART I
About the Ingredients

1

The History of Unix, GNU, and Linux

The Unix tradition has a long history, and Linux comes from the Unix tradition, so to understand Linux one must understand Unix and to understand Unix one must understand its history. Before Unix, a developer would submit a stack of punched cards, each card representing a command, or part of a command. These cards would be read and executed sequentially by the computer. The developer would receive the generated output after the job had completed. This would often be a few days after the job had been submitted; if there was an error in the code, the output was just the error and the developer had to start again. Later, teletype and various forms of timesharing systems sped up the matter considerably, but the model was basically the same: a sequence of characters (punch cards, or keys on keyboards — it's still just a string of characters) submitted as a batch job to be run (or fail to run), and for the result to come back accordingly. This is significant today in that it is still how data is transmitted on any computerized system — it's all sequences of characters, transmitted in order. Whether a text file, a web page, a movie, or music, it is all just strings of ones and zeroes, same as it ever was. Anything that looks even slightly different is simply putting an interface over the top of a string of ones and zeroes.

Unix and various other interactive and timesharing systems came along in the mid-1960s. Unix and its conventions continue to be central to computing practices today; its influences can be seen in DOS, Linux, Mac OS X, and even Microsoft Windows.

UNIX

In 1965, Bell Labs and GE joined a Massachusetts Institute of Technology (MIT) project known as MULTICS, the Multiplexed Information and Computing System. Multics was intended to be a stable, timesharing OS. The "Multiplexed" aspect added unnecessary complexity, which eventually led Bell Labs to abandon the project in 1969. Ken Thompson, Dennis Ritchie, Doug McIlroy, and Joe Ossanna retained some of the ideas behind it, took out a lot of the complexity, and came up with Unix (a play on the word MULTICS, as this was a simplified operating system inspired by MULTICS).

An early feature of Unix was the introduction of *pipes* — something that Doug McIlroy had been thinking about for a few years and was implemented in Unix by Ken Thompson. Again, it took the same notion of streamed serial data, but pipes introduced the idea of having `stdin` and `stdout`, through which the data would flow. Similar things had been done before, and the concept is fairly simple: One process creates output, which becomes input to another command. The Unix pipes method introduced a concept that dramatically affected the design of the rest of the system.

Most commands have a file argument as well, but existing commands were modified to default to read from their "Standard Input" (`stdin`) and "Standard Output" (`stdout`); the pipe can then "stream" the data from one tool to another. This was a novel concept, and one that strongly defines the Unix shell; it makes the whole system a set of generically useful tools, as opposed to monolithic, single-purpose applications. This has been summarized as "do one thing and do it well." The GNU toolchain was written to replace Unix while maintaining compatibility with Unix tools. The developers on the GNU project often took the opportunity presented by rewriting the tool to include additional functionality, while still sticking to the "do one thing and do it well" philosophy.

The GNU project was started in 1983 by Richard Stallman, with the intention of replacing proprietary commercial Unices with Free Software alternatives. GNU had all but completed the task of replacing all of the userspace tools by the time the Linux kernel project started in 1991. In fact, the GNU tools generally perform the same task at least as well as their original Unix equivalents, often providing extra useful features borne of experience in the real world. Independent testing has shown that GNU tools can actually be more reliable than their traditional Unix equivalents (`http://www.gnu.org/software/reliability.html`*).*

For example, the `who` command lists who is logged in to the system, one line per logged-in session. The `wc` command counts characters, words, and lines. Therefore, the following code will tell you how many people are logged in:

```
who | wc -l
```

There is no need for the `who` tool to have an option to count the logged-in users because the generic `wc` tool can do that already. This saves some small effort in `who`, but when that is applied across the whole range of tools, including any new tools that might be written, a lot of effort and therefore complexity, which means a greater likelihood of the introduction of additional bugs, is avoided. When this is applied to more complicated tools, such as `grep` or even `more`, the flexibility of the system is increased with every added tool.

In the case of `more`*, this is actually more tricky than it seems; first it has to find out how many columns and rows are available. Again, there is a set of tools that combine to provide this information. In this way, every tool in the chain can be used by the other tools.*

Also this system means that you do not have to learn how each individual utility implements its "word count" feature. There are a few defacto standard switches; `-q` typically means Quiet, `-v` typically means Verbose, and so on, but if `who -c` meant "count the number of entries," then `cut -c <n>`, which means "cut the first *n* characters," would be inconsistent. It is better that each tool does its own job, and that `wc` do the counting for all of them.

For a more involved example, the `sort` utility just sorts text. It can sort alphabetically or numerically (the difference being that "10" comes before "9" alphabetically, but after it when sorted numerically), but it doesn't search for content or display a page at a time. `grep` and `more` can be combined with `sort` to achieve this in a pipeline:

```
grep foo /path/to/file | sort -n -k 3 | more
```

This pipeline will search for `foo` in `/path/to/file`. The output (`stdout`) from that command will then be fed into the `stdin` of the `sort` command. Imagine a garden hose, taking the output from `grep` and attaching it to the input for `sort`. The `sort` utility takes the filtered list from `grep` and outputs the sorted results into the stdin of `more`, which reads the filtered and sorted data and paginates it.

It is useful to understand exactly what happens here; it is the opposite of what one might intuitively assume. First, the `more` tool is started. Its input is attached to a pipe. Then `sort` is started, and its output is attached to that pipe. A second pipe is created, and the `stdin` for `sort` is attached to that. `grep` is then run, with its `stdout` attached to the pipe that will link it to the `sort` process.

When `grep` begins running and outputting data, that data gets fed down the pipe into `sort`, which sorts its input and outputs down the pipe to `more`, which paginates the whole thing. This can affect what happens in case of an error; if you mistype "`more`," then nothing will happen. If you mistype "`grep`," then `more` and `sort` will have been started by the time the error is detected. In this example, that does not matter, but if commands further down the pipeline have some kind of permanent effect (say, if they create or modify a file), then the state of the system will have changed, even though the whole pipeline was never executed.

"Everything Is a File" and Pipelines

There are a few more key concepts that grew into Unix as well. One is the famous "everything is a file" design, whereby device drivers, directories, system configuration, kernel parameters, and processes are all represented as files on the filesystem. Everything, whether a plain-text file (for example, `/etc/hosts`), a block or character special device driver (for example, `/dev/sda`), or kernel state and configuration (for example, `/proc/cpuinfo`) is represented as a file.

The existence of pipes leads to a system whereby tools are written to assume that they will be handling streams of text, and indeed, most of the system configuration is in text form also. Configuration files can be sorted, searched, reformatted, even differentiated and recombined, all using existing tools.

The "everything is a file" concept and the four operations (`open`, `close`, `read`, `write`) that are available on the file mean that Unix provides a really clean, simple system design. Shell scripts themselves are another example of a system utility that is also text. It means that you can write programs like this:

```
#!/bin/sh
cat $0
echo "==="
tac $0
```

This code uses the `cat` facility, which simply outputs a file, and the `tac` tool, which does the same but reverses it. (The name is therefore quite a literal interpretation of what the tool does, and quite a typical example of Unix humor.) The variable `$0` is a special variable, defined by the system, and contains the name of the currently running program, as it was called.

So the output of this command is as follows:

```
#!/bin/sh
cat $0
echo "==="
tac $0
===
tac $0
echo "==="
cat $0
#!/bin/sh
```

The first four lines are the result of `cat`, the fifth line is the result of the `echo` statement, and the final four lines are the output of `tac`.

BSD

AT&T/Bell Labs couldn't sell Unix because it was a telecommunications monopoly, and as such was barred from extending into other industries, such as computing. So instead, AT&T gave Unix away, particularly to universities, which were naturally keen to get an operating system at no cost. The fact that the schools could also get the source code was an extra benefit, particularly for administrators but also for the students. Not only could users and administrators run the OS, they could see (and modify) the code that made it work. Providing access to the source code was an easy choice for AT&T; they were not (at that stage) particularly interested in developing and supporting it themselves, and this way users could support themselves. The end result was that many university graduates came into the industry with Unix experience, so when they needed an OS for work, they suggested Unix. The use of Unix thus spread because of its popularity with users, who liked its clean design, and because of the way it happened to be distributed.

Although it was often given away at no cost or low cost and included the source code, Unix was not Free Software according to the Free Software Foundation's definition, which is about freedom, not cost. The Unix license prohibited redistribution of Unix to others, although many users developed their own patches, and some of those shared patches with fellow Unix licensees. (The patches would be useless to someone who didn't already have a Unix license from AT&T. The core software was still Unix; any patches were simply modifications to that.) Berkeley Software Distribution (BSD) of the University of California at Berkeley created and distributed many such patches, fixing bugs, adding features, and just generally improving Unix. The terms "Free Software" and "Open Source" would not exist for a long time to come, but all this was distributed on the understanding that if something is useful, then it may as well be shared. TCP/IP, the two core protocols of the Internet, came into Unix via BSD, as did BIND, the DNS (Domain Name System) server, and the Sendmail MTA (mail transport agent). Eventually, BSD developed so many patches to Unix that the project had replaced virtually all of the original Unix source code. After a lawsuit, AT&T and BSD made peace and agreed that the few remaining AT&T components of BSD would be rewritten or relicensed so that BSD was not the property of AT&T, and could be distributed in its own right. BSD has since forked into NetBSD, OpenBSD, FreeBSD, and other variants.

GNU

As mentioned previously, the GNU project was started in 1983 as a response to the closed source software that was by then being distributed by most computer manufacturers along with their hardware. Previously, there had generally been a community that would share source code among users, such that if anyone felt that an improvement could be made, they were free to fix the code to work as they would like. This hadn't been enshrined in any legally binding paperwork; it was simply the culture in which developers naturally operated. If someone expressed an interest in a piece of software, why would you not give him a copy of it (usually in source code form, so that he could modify it to work on his system? Very few installations at the time were sufficiently similar to assume that a binary compiled on one machine would run on another). As Stallman likes to point out, "Sharing of software…is as old as computers, just as sharing of recipes is as old as cooking."[1]

Stallman had been working on the Incompatible Timesharing System (ITS) with other developers at MIT through the 1970s and early 1980s. As that generation of hardware died out, newer hardware came out, and — as the industry was developing and adding features — these new machines came with bespoke operating systems. Operating systems, at the time, were usually very hardware-specific, so ITS and CTSS died as the hardware they ran on were replaced by newer designs.

ITS was a pun on IBM's Compatible Time Sharing System (CTSS), which was also developed at MIT around the same time. The "C" in CTSS highlighted the fact that it was somewhat compatible with older IBM mainframes. By including "Incompatible" in its name, ITS gloried in its rebellious incompatibility.

Stallman's turning point occurred when he wanted to fix a printer driver, such that when the printer jammed (which it often did), it would alert the user who had submitted the job, so that she could fix the jam. The printer would then be available for everyone else to use. The user whose job had jammed the printer wouldn't get her output until the problem was fixed, but the users who had submitted subsequent jobs would have to wait even longer. The frustration of submitting a print job, then waiting a few hours (printers were much slower then), only to discover that the printer had already stalled before you had even submitted your own print job, was too much for the users at MIT, so Stallman wanted to fix the code. He didn't expect the original developers to work on this particular feature for him; he was happy to make the changes himself, so he asked the developers for a copy of the source code. He was refused, as the driver software contained proprietary information about how the printer worked, which could be valuable competitive information to other printer manufacturers.

What offended Stallman was not the feature itself, it was that one developer was refusing to share code with another developer. That attitude was foreign to Stallman, who had taken sharing of code for granted until that stage. The problem was that the software — in particular the printer driver — was not as free (it didn't convey the same freedoms) as previous operating systems that Stallman had worked with. This problem prevailed across the industry; it was not specific to one particular platform, so changing hardware would not fix the problem.

[1] *Free Software, Free Society*, 2002, Chapter 1. ISBN 1-882114-98-1

GNU stands for "GNU's Not Unix," which is a recursive acronym; if you expand the acronym "IBM," you get "International Business Machines," and you're done. If you expand "GNU," you get "GNU's Not Unix's Not Unix." Expand that, and you get "GNU's Not Unix's Not Unix's Not Unix" and so on. This is an example of "hacker humor," which is usually quite a dry sense of humor, with something a little bit clever or out of the ordinary about it. At the bottom of the `grep` *manpage, under the section heading "NOTES" is a comment: "GNU's not Unix, but Unix is a beast; its plural form is Unixen," a friendly dig at Unix.*

Richard Stallman is a strong-willed character (he has described himself as "borderline autistic"), with a very logical mind, and he determined to fix the problem in the only way he knew how: by making a new operating system that would maintain the old unwritten freedoms to allow equal access to the system, including the code that makes it run. As no such thing existed at the time, he would have to write it. So he did.

STALLMAN CHARGES AHEAD!

From CSvax:pur-ee:inuxc!ixn5c!ihnp4!houxm!mhuxi!eagle!mit-vax!mit-eddie!RMS@MIT-OZ

Newsgroups: net.unix-wizards,net.usoft

Organization: MIT AI Lab, Cambridge, MA

From: RMS%MIT-OZ@mit-eddie

Subject: new Unix implementation

Date: Tue, 27-Sep-83 12:35:59 EST

Free Unix!

Starting this Thanksgiving I am going to write a complete Unix-compatible software system called GNU (for Gnu's Not Unix), and give it away free to everyone who can use it. Contributions of time, money, programs and equipment are greatly needed.

To begin with, GNU will be a kernel plus all the utilities needed to write and run C programs: editor, shell, C compiler, linker, assembler, and a few other things. After this we will add a text formatter, a YACC, an Empire game, a spreadsheet, and hundreds of other things. We hope to supply, eventually, everything useful that normally comes with a Unix system, and anything else useful, including on-line and hardcopy documentation.

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 filenames, file version

numbers, a crashproof file system, filename completion perhaps, terminal-independent display support, and 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 have network software based on MIT's chaosnet protocol, far superior to UUCP. We may also have something compatible with UUCP.

Who Am I?

I am Richard Stallman, inventor of the original much-imitated EMACS editor, now at the Artificial Intelligence Lab at MIT. I have worked extensively on compilers, editors, debuggers, command interpreters, the Incompatible Timesharing System and the Lisp Machine operating system. I pioneered terminal-independent display support in ITS. In addition I have implemented one crashproof file system and two window systems for Lisp machines.

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. I cannot in good conscience sign a nondisclosure agreement or a software license agreement.

So that I can continue to use computers without violating my principles, 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.

How You Can Contribute

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

One computer manufacturer has already offered to provide a machine. But we could use more. One consequence you can expect if you donate machines is that GNU will run on them at an early date. The machine had better be able to operate in a residential area, and not require sophisticated cooling or power.

Individual programmers can contribute by writing a compatible duplicate of some Unix utility and giving it to me. 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. Most interface specifications are fixed by Unix compatibility. If each contribution works with the rest of Unix, it will probably work with the rest of GNU.

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, but I'm looking for people for whom knowing they are helping humanity is as important as 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.

For more information, contact me.

Unix already existed, was quite mature, and was nicely modular. So the GNU project was started with the goal of replacing the userland tools of Unix with Free Software equivalents. The kernel was another part of the overall goal, although one can't have a kernel in isolation — the kernel needs an editor, a compiler, and a linker to be built, and some kind of initialization process in order to boot. So existing proprietary software systems were used to assemble a free ecosystem sufficient to further develop itself, and ultimately to compile a kernel. This subject had not been ignored; the Mach microkernel had been selected in line with the latest thinking on operating system kernel design, and the HURD kernel has been available for quite some time, although it has been overtaken by a newer upstart kernel, which was also developed under, and can also work with, the GNU tools.

HURD is "Hird of Unix-Replacing Daemons," because its microkernel approach uses multiple userspace background processes (known as daemons in the Unix tradition) to achieve what the Unix kernel does in one monolithic kernel. HIRD in turn stands for "Hurd of Interfaces Representing Depth." This is again a recursive acronym, like GNU ("GNU's Not Unix") but this time it is a pair of mutually recursive acronyms. It is also a play on the word "herd," the collective noun for Gnus.

As the unwritten understandings had failed, Stallman would need to create a novel way to ensure that freely distributable software remained that way. The GNU General Public License (GPL) provided that in a typically intelligent style. The GPL uses copyright to ensure that the license itself cannot be changed; the rest of the license then states that the recipient has full right to the code, so long as he grants the same rights to anybody he distributes it to (whether modified or not) and the license does not change. In that way, all developers (and users) are on a level playing field, where the code is effectively owned by all involved, but no one can change the license, which ensures that equality. The creator of a piece of software may dual-license it, under the GPL and a more restrictive license; this has been done many times — for example, by the MySQL project.

One of the tasks taken on by the GNU project was — of course — to write a shell interpreter as free software. Brian Fox wrote the bash (Bourne Again SHell) shell — its name comes from the fact that the original `/bin/sh` was written by Steve Bourne, and is known as the Bourne Shell. As bash takes the features of the Bourne shell, and adds new features, too, bash is, obviously, the Bourne Again Shell. Brian also wrote the readline utility, which offers flexible editing of input lines of text before submitting them for parsing. This is probably the most significant feature to make bash a great interactive shell. Brian Fox was the first employee of the Free Software Foundation, the entity set up to coordinate the GNU project.

You've probably spotted the pattern by now; although bash isn't a recursive acronym, its name is a play on the fact that it's based on the Bourne shell. It also implies that bash is an improvement on the original Bourne shell, in having been "bourne again."

LINUX

Linus Torvalds, a Finnish university student, was using Minix, a simple Unix clone written by Vrije Universiteit (Amsterdam) lecturer Andrew Tanenbaum, but Torvalds was frustrated by its lack of features and the fact that it did not make full use of the (still relatively new) Intel 80386 processor, and in particular its "protected mode," which allows for much better separation between the kernel and userspace. Relatively quickly, he got a working shell, and then got GCC, the GNU C compiler (now known as the GNU Compiler Collection, as it has been extended to compile various flavors of C, Fortran, Java, and Ada) working. At that stage, the kernel plus shell plus compiler was enough to be able to "bootstrap" the system — it could be used to build a copy of itself.

TORVALDS' NEWSGROUP POST

On August 25, 1991, Torvalds posted the following to the MINIX newsgroup `comp.os.minix`:

From: torvalds@klaava.helsinki.fi (Linus Benedict Torvalds)
To: Newsgroups: comp.os.minix
Subject: What would you like to see most in minix?
Summary: small poll for my new operating system

Hello everybody out there using minix-

I'm doing a (free) operating system (just a hobby, won't be big and professional like gnu) for 386 (486) AT clones. This has been brewing since april, and is starting to get ready. I'd like any feedback on things people like/dislike in minix, as my OS resembles it somewhat (same physical layout of the file-sytem due to practical reasons) among other things.

I've currently ported bash (1.08) an gcc (1.40), and things seem to work. This implies that i'll get something practical within a few months, and I'd like to know what features most people want.

Any suggestions are welcome, but I won't promise I'll implement them :-)

Linus Torvalds torvalds@kruuna.helsinki.fi

What is interesting is that Torvalds took the GNU project's inevitable success for granted; it had been going for eight years, and had basically implemented most of its goals (bar the kernel). Torvalds also, after initially making the mistake of trying to write his own license (generally inadvisable for those of us who are not experts in the minutiae of international application of intellectual property law), licensed the kernel under the GNU GPL (version 2) as a natural license for the project.

In practice, this book is far more about shell scripting with Unix and GNU tools than specifically about shell scripting under the Linux kernel; in general, the majority of the tools referred to are GNU tools from the Free Software Foundation: `grep`, `ls`, `find`, `less`, `sed`, `awk`, `bash` itself of course, `diff`, `basename`, and `dirname`; most of the critical commands for shell scripting on Linux

are GNU tools. As such, some people prefer to use the phrase "GNU/Linux" to describe the combination of GNU userspace plus Linux kernel. For the purposes of this book, the goal is to be technically accurate while avoiding overly political zeal. RedHat Linux is what RedHat calls its distribution, so it is referred to as RedHat Linux. Debian GNU/Linux prefers to acknowledge the GNU content so we will, too, when referring specifically to Debian. When talking about the Linux kernel, we will say "Linux"; when talking about a GNU tool we will name it as such. Journalists desperate for headlines can occasionally dream up a far greater rift than actually exists in the community. Like any large family, it has its disagreements — often loudly and in public — but we will try not to stoke the fire here.

Unix was designed with the assumption that it would be operated by engineers; that if somebody wanted to achieve anything with it, he or she would be prepared to learn how the system works and how to manipulate it. The elegant simplicity of the overall design ("everything is a file," "do one thing and do it well," etc.) means that principles learned in one part of the system can be applied to other parts.

The rise in popularity of GNU/Linux systems, and in particular, their relatively widespread use on desktop PCs and laptop systems — not just servers humming away to themselves in dark datacenters — has brought a new generation to a set of tools built on this shared philosophy, but without necessarily bringing the context of history into the equation.

Microsoft Windows has a very different philosophy: The end users need not concern themselves with how the underlying system works, and as a result, should not expect it to be discernable, even to an experienced professional, because of the closed-source license of the software. This is not a difference in quality or even quantity; this is a different approach, which assumes a hierarchy whereby the developers know everything and the users need know nothing.

As a result, many experienced Windows users have reviewed a GNU/Linux distribution and found to their disappointment that to get something configured as it "obviously" should be done, they had to edit a text file by hand, or specify a certain parameter. This flexibility is actually a strength of the system, not a weakness. In the Windows model, the user does not have to learn because they are not allowed to make any decisions of importance: which kernel scheduler, which filesystem, which window manager. These decisions have all been made to a "one size fits most" level by the developers.

SUMMARY

Although it is quite possible to administer and write shell scripts for a GNU/Linux system without knowing any of the history behind it, a lot of apparent quirks will not make sense without some appreciation of how things came to be the way they are. There is a difference between scripting for a typical Linux distribution, such as RedHat, SuSE, or Ubuntu, and scripting for an embedded device, which is more likely to be running `busybox` than a full GNU set of tools. Scripting for commercial Unix is slightly different again, and much as a web developer has to take care to ensure that a website works

in multiple browsers on multiple platforms, a certain amount of testing is required to write solid cross-platform shell scripts.

Even when writing for a typical Linux distribution, it is useful to know what is where, and how it came to be there. Is there an `/etc/sysconfig`? Are init scripts in `/etc/rc.d/init.d` or `/etc/init.d`, or do they even exist in that way? What features can be identified to see what tradition is being followed by this particular distribution? Knowing the history of the system helps one to understand whether the syntax is `tar xzf` or `tar -xzf`; whether to use `/etc/fstab` or `/etc/vfstab`; whether running `killall httpd` will stop just your Apache processes (as it would under GNU/Linux) or halt the entire system (as it would on Solaris)!

The next chapter follows on from this checkered history to compare the variety of choices available when selecting a Unix or GNU/Linux environment.