Book Reviews

MARK LAMOURINE AND RIK FARROW

BPF Performance Tools: Linux System and Application Observability
Brendan Gregg
Addison-Wesley Professional, 2019, 880 pages
ISBN 978-0-13-655482-0

Reviewed by Mark Lamourine

I haven't finished thoroughly reading *BPF Performance Tools*. I'm not sure I will ever touch and try everything that Gregg offers in this 880-page book. Typically when I read a computer technology book, I have some hooks into the topic to start. I can skim through once and then choose a few sections to dive deeply into and get a good sense of what the book is about and how it will read for different audiences. Opening and scanning this book felt like stepping through a door marked "Authorized Personnel Only" into a control room for a nuclear power plant or a SpaceX launch.

BPF and the BCC tools based on BPF provide visibility into the operation of the Linux kernel and subsystems. Formally, the current name is Extended Berkeley Packet Filters, but Gregg indicates that most people just call it BPF. BPF is nominally about the performance of apps run by a Linux kernel, but it is not limited to tuning. As Gregg presents it, BPF is much more a diagnostic tool.

The cover of *BPF Performance Tools* contains an image that is indicative of the depth and range of the capabilities of the tool set. The image shows dozens of targeted scripts that give visibility into every part of the Linux environment. All of this is made possible by the BPF virtual machine and the probes embedded in each of the kernel components. From the user perspective, BPF and BCC themselves are fairly simple, but the vista they open up can be overwhelming.

Most sysadmins can go a lifetime with only a cursory understanding of the deep internal workings of the Linux kernel. That's as intended. If you needed to be able to trace the flow of blocks of data from disk sectors or an SSD though the kernel to a string printed out on the CLI just to write "hello world," very little else would get done. Occasionally, though, we see problems or unexpected behaviors and interactions as the system runs, and then we need to look underneath to see what the system is actually doing.

Such a significant but generally invisible subsystem needs some introduction. The first five chapters introduce the technology that makes up the BPF mechanism and the suite of tools that use it. This only makes up the first fifth of the book, but it fills 200 pages. There are two major tool sets: BCC, a set of Python scripts that run common operations, and `bpftrace`, a program that can run one-liner probes. Each gets a chapter of its own. With that introduction done Gregg can begin showing how to use BPF to probe each of the subsystems of a running Linux machine.

In the main body of the book, Gregg steps through the boxes in the cover illustration. The CPU, memory, disk I/O, and networking chapters make up the parts of a bare metal machine, but BPF probes don't stop there. There are chapters on profiling programs and scripts in various languages and on monitoring VMs and containers. Gregg doesn't limit himself to BPF probes either. In each chapter, he includes first the traditional tools that already existed. He shows what they are capable of and how they are used and then moves on to how to use BPF probes to learn more.

The book concludes with chapters on common tips and tricks and on reusable BPF tool one-liners and sample runs of each of the tools with annotated output.

There is a lot here to digest and it concerns what a novice would find to be absolute arcana. That's not to say it's beyond the use of a range of sysadmins from junior to architect to forensic analyst. I've often found that by skimming a topic I can learn enough so that when a problem arises related to the topic, I remember and can return for more depth as needed. This isn't a cover-to-cover book. There is no narrative progression. A reader will do best to go straight to the topic they need and begin using it immediately. BPF is a diagnostic tool, so each use will lead to new queries until the user comes to understand the behavior of the system they are examining.

BPF offers a great tool set for understanding not just broken systems but well run ones. Diagnostic profiling often depends on first establishing a baseline of normality. A reader who wants to deeply understand the normal operation of a Linux system could do worse than to experiment with BPF on the systems they have, using it as a flashlight in the dark caves underneath the shell and GUI.

Ubuntu Unleashed 2019 Edition

Matthew Helmke
Addison-Wesley Professional, 2019, 800 pages
ISBN: 978-0-13-498546-6

Reviewed by Mark Lamourine

I've worked almost exclusively on Red Hat and Fedora Linux systems for more than a decade. Prior to that I ran my home systems on Ubuntu for several years. Recently I started work at a place that uses RPM- and DEB-based systems side by side, and I thought it would be good to get a refresher on modern Ubuntu.

When you include the year in the name of a book, you know it will have a limited life, but in technology today that's pretty much a given. One thing I was curious about when I selected *Ubuntu Unleashed* was how much would be familiar to me from my Edgy and Feisty days. I was also interested in seeing how much of Ubuntu was just Linux as I already knew it. It turns out that much of what you find in an encyclopedic volume like this ages better than you might expect.

The first edition of *Ubuntu Unleashed*, published in 2006, was written by Paul and Andrew Hudson, and they still get credit on the inside cover page. There have been near-annual updates since then.

Ubuntu Unleashed feels like a very big, shallow wading pool. It has a paragraph or two on nearly everything to do with a modern Linux operating system. It's not useful as a tutorial for a completely new user or as a reference for a master. It is very well suited to a novice with some experience or for an expert from a different distribution. In both of these cases, the reader will have some context to use but will have gaps that need filling.

In each section, Helmke introduces the topic, defining terms and giving context about why it is important and where it fits into the OS. He only touches lightly on each point before moving on, however, and each chapter closes with a list of books and websites for deeper study. Another way to think of a book like this is as an annotated index to some larger compendium of knowledge.

I did find several things that raised an eyebrow. I am an Emacs user for development work and use `vi` for single file edits. I started using Emacs before there was a GUI for it. I was surprised though to see even a reference to Emacs as an editor option and even more because it was listed first. I would never recommend Emacs to a new user. `vim` has become as capable a text editor as Emacs ever was, and the community to learn from is much larger. I would advise against ever invoking Emacs on a single file as Helmke does. I understand wanting to avoid getting involved in the editor wars, but I think in some things it is acceptable for an author to have opinions. Later, the four-page section on KVM followed by a page for VirtualBox and a paragraph each for VMware and Xen shows that he does make use of his editorial prerogative.

Another thing that was curious to me was the treatment of the boot process and of `init` systems. It makes sense to continue to treat legacy `init` systems as well as `upstart` and `systemd`, as there will be readers who must work on older systems. The problem here is that the discussions of the different systems is interlaced in a way that I find confusing, and I am familiar with all three. I would have preferred a general discussion of the bootstrap process and then a distinct section for each boot method, treating how the user can view and interact with it.

That said, my personal weak points are in kernel tuning and module management. A quick pass over those sections gave me a number of tips to follow up to start filling in the gaps, with references to more detail when I find the time.

The table of contents of the book concludes with three bonus chapters that are available on the publisher's web site. These are short topics in downloadable PDF on Perl, PHP, and Python. Again, I'm a little surprised to see the first two, but they make sense for completeness' sake. There are also PDFs with updates specific to Ubuntu versions that were released or updated after the manuscript went to print.

Ubuntu Unleashed 2019 Edition lives up to the author's goals to provide a resource for "those wanting to become intermediate or advanced users." It is a touchstone that you can use to find direction and move on when learning about the whole range of tasks on a modern Ubuntu system.

Programming with Types: Examples in Typescript

Vlad Riscutia
Manning Publications, 2020, 336 pages
ISBN 978-1-61-729641-3

Reviewed by Mark Lamourine

It took me a while to figure out where to put *Programming with Types* on my bookshelf. The other books I have read recently tend to fit either on the programming language or cloud technology shelves. Initially, I thought that it would sit next to my other Typescript and web programming books, but it became clear quickly that Typescript was really incidental to the content. *Programming with Types* is really more about technique than technology. It would not be out of place in an undergraduate software engineering course.

Most books about imperative programming languages focus on syntax and logical controls: conditionals, branching, iteration, recursion, and the logical structures that the language presents to implement them. Types and structures are presented as merely a way to represent and manipulate data, but

are subservient to the algorithm. Riscutia inverts the emphasis, putting the data types up front and choosing the best algorithmic techniques to suit the data.

The author presents types and strong typing as tools to prevent errors and make the intent of the code clear to the reader. He goes as far as calling the use of primitive numerical types without semantic typing an anti-pattern called "primitive obsession." He claims that errors such as the Mars Climate Orbiter error that caused the spacecraft to disintegrate in the Martian atmosphere might have been prevented if the coders had used numeric subtypes that indicated the unit. If, instead of float, they had used subtypes Newton seconds and pound-force seconds, the mismatch would have been caught by the compiler and would have highlighted the need for a conversion function to make the two sets of routines interact properly. While I agree that more rigor in general coding practice would be a good thing, I'm not sure I would go as far as calling the use of bare numeric types an anti-pattern.

It is clear that Riscutia is conversant and interested in the theory of typing and has the mathematical and logical rigor that good strong typing requires. Weak type systems can make for quick efficient coding, but they are, by design, prone to and even accepting of the kinds of errors that can result. Writing well-designed, strongly typed systems requires the coder to consider carefully the signature of every function and, at times, to do extra work to account for algorithms that are identical in all ways but that they operate on trivially different types. Generic type constructs exist precisely to address this but can be difficult to conceptualize and define well.

The author expects the reader to be at least conversant with all of the techniques and styles that he addresses. He doesn't try to teach functional or object-oriented programming, or even class definition and structure composition. He is entirely devoted to understanding and managing the data relationships. He dips regularly into theory but not deeply. Advanced techniques such as closures and promises get only a paragraph of exposition before he begins to show how to use them and how they will respond.

In each chapter, Riscutia focuses on a coding technique that you would find in a number of other books. He doesn't advocate one style over another. He starts, as you would expect, with primitive types and then goes on to cover collections like arrays and structures. There is a chapter on object orientation and one on functional programming. Another talks about the type constructs and techniques of meta-programming. The emphasis is on using appropriate data types and using them in effective ways. This change in perspective highlights the importance of properly modeling the data in a way that I found interesting and enlightening. It is easy to let the programming language features and the algorithms drive a design, but in the end it is the data that defines the job.

Programming with Types is a fresh breeze for an experienced generalist software developer like myself. It is a welcome change and may find its place next to some of the classics on my shelf.

UNIX: A History and a Memoir

Brian Kernighan
Kindle Direct Publishing, 2020, 183 pages
ISBN 978-1-695-97855-3

Reviewed by Rik Farrow

Brian Kernighan has written or co-authored many books over the years, but this one is different. Using a conversational style, Brian tells the story of UNIX—not just the operating system and its core utilities, but the environment it grew in and the people involved.

The *memoir* as part of the title is accurate, as this is the viewpoint of an insider at Bell Labs in Murray Hill, working surrounded by the people who created not just UNIX, but C, C++, *roff, Programmer's Workbench and Writer's Workbench, `yacc`, `lex`, `awk`, and countless other tools. As someone who *needed* to know about UNIX in 1978 but didn't encounter UNIX for another five years, I found myself endlessly curious about not just the operating system but the philosophy that obviously influenced it.

Part of that overall attitude showed in the early man pages, succinct with a hint of dry humor, when all anyone had *were* the man pages and USENIX meetings, as the Internet didn't exist and there were *no* technical books other than manuals and textbooks. Brian explains that the man pages were largely the work of Dennis Ritchie and Doug McIlroy, and it's their personalities that provide the style found in the UNIX man pages that is so difficult to mimic.

Brian came to Bell Labs as a programmer, with the majority of his focus on publishing. He explains how crucial the ability to produce technical reports, papers, and books was to the survival of UNIX in the first decade. He has already described the minimal capabilities of the PDP 7 when telling of the writing of UNIX by Ken Thompson, but I think we tend to forget that everything was terribly primitive at the time, including the ability to typeset technical documents. A high-end digital printer of that era was a line printer that was ASCII-only. The ability to typeset equations and diagrams was an enormous advance, and one that Brian participated in by writing `eqn` and `pic`.

I enjoyed reading Brian's tales, learning something about the personalities of people I mostly knew by their creations. I do wonder how many others will be as taken as I was by the histories, as they grew up in an era where information is a quick online search away. But reading the first-hand accounts dispelled many of the myths surrounding the birth of UNIX and its associated

parts. And we can only dream of being able to work at a place like Bell Labs back in the days when the telephone monopoly could afford to lavish resources on pure research and hiring prodigies.

There are weaknesses in a book like this one, in that Brian's focus is Bell Labs in Murray Hill, New Jersey. He mentions USENIX, but focuses on its role in expanding Netnews, something that the various Labs actually supported, via providing dial-up long distance connections used for UUCP mail and Netnews. Brian talks about the importance of the AT&T lawsuit in 1989 surrounding the BSD UNIX implementation, but misses that the lawsuit was against BSDi and the Regents of the University of California. I found his explanation of the UNIX file system a better match for NTFS, but that's merely a quibble alongside his other revelations.

Will we ever see the day where another Bell Labs-style incubator exists? For a while I thought that Google might be that place, even as Murray Hill programmers wound up working there (Thompson, Pike, and Presotto, for example). Today, I believe we need to look elsewhere, or give up on expecting another monopoly corporation to behave in a manner that benefits the public more than its shareholders.