Book Reviews

RIK FARROW AND MARK LAMOURINE

The Design and Implementation of the FreeBSD Operating System (2nd Edition)

Marshall Kirk McKusick, George V. Neville-Neil, and Robert N. M. Watson
Addison-Wesley Professional, 2014; 928 pages
ISBN 978-0-321-96897-5

Reviewed by Rik Farrow

This book comes out of a lineage of books about the BSD operating system, starting with *The Design and Implementation of 4.3BSD UNIX* in 1989. While its focus on FreeBSD sets this book apart from other operating systems books, where the focus is Linux, that's not all that sets it apart.

Kirk McKusick has been involved in key design decisions that still have bearing on UNIX-related systems since he was a graduate student sharing an office with Bill Joy. And this book reflects not only McKusick's influence on the designs of file systems and virtual-memory systems, but also that of its two other authors.

Whereas a book like Robert Love's *Linux Kernel Development* dives into getting, building, and examining kernel code, *Design and Implementation* stays at a higher level. Algorithms and data structures are explained, but so are the design decisions behind *why* a particular algorithm or design was chosen.

Soft updates provide a particularly contentious example. Early Linux file systems could create and delete files much faster than the 4.3 BSD fast file system (FFS), because the authors of ext2 had decided to do away with ordered, synchronous writes of file-system metadata. The FreeBSD developers' response, led by McKusick, was to create a process called *soft updates*, which allows metadata updates to occur asynchronously, but still in an ordered manner. In the Linux world, soft updates are spurned as too complicated. In this book, they are explained in clear and concise text, both why they are considered necessary and how they need to work. Approaches that log metadata updates are considered in the following section (the approach used in ext3).

Like operating systems books in general, the book begins with a history of UNIX (but written by one of its participants), followed by an overview of the kernel. Process management follows, then a completely rewritten chapter on security. If you are seriously interested in operating system security features, this chapter provides an excellent overview of the many mechanisms that have appeared, and been implemented, over the past 25 years. While the Linux security module and the related SELinux and type enforcement get only brief mention, there are thorough discussions of access control lists, mandatory access control, the new NFSv4 ACLs, security event auditing, cryptographic services, random number generator, jails, and the Capsicum capabilities model—a recent addition to FreeBSD.

The next chapter, on memory management, is just as long as the security chapter, and just as detailed. The next part of the book covers the I/O system, starting with overview, then devices in general, moving to FFS, then a new chapter on the Zettabyte File System. Again, this chapter would be useful to anyone who wants a deep understanding of ZFS, whether you are using FreeBSD, Linux, or Solaris descendants like illumos. The I/O section ends with a chapter on NFS, including NFSv4.

Part four covers Interprocess Communication, which begins with IPC and continues with chapters on network layer protocols, like IPv4 and IPv6, and transport layer protocols. The book concludes with a chapter on system startup and shutdown and a glossary.

Each chapter ends with exercises and one or more pages of references. The exercises cover ideas from each chapter and help the dedicated reader to think about potential solutions that go beyond what's covered in each chapter.

I did what I usually do with large technical books: I jumped around, after reading all of the introductory material, focusing on the parts I found most interesting. The writing makes this easy to do, in that I rarely found myself referred to another section in the book. This is not unlike the design of FreeBSD itself, which tends to be more modular than Linux.

In a world, especially an OS research world, dominated by Linux, you might really wonder why you would take the time to read a book on FreeBSD. The real reason is that there is a wealth of experience, a record of different approaches taken, written by three FreeBSD committers, all with stellar records. It would be a shame to miss out on all of this knowledge because of parochialism.

Think Bayes

Allen B. Downey
O'Reilly Media, Inc., 2013; 190 pages
ISBN 978-1-449-37078-7

Reviewed by Mark Lamourine

If you've read my previous reviews of Allen Downey's books, you'll know I'm a fan. His first three books covered Python programming, statistics, and complexity. His most recent is a practical exploration of Bayesian statistics, and I like this one as well.

Downey's purpose in all of his books is to set the reader on an exploration of the topic rather than to sit her down in a lecture hall. In each chapter or section, he introduces a real-world problem and then shows the reader the toolbox that will be needed to solve it. This usually includes external references to more in-depth treatments and often to primary source online data sets. His tone and style are very easy going, but this sometimes belies the difficulty and significance of the subject matter.

In *Think Bayes*, as in his other books, Downey aims to achieve something that might seem oxymoronic: applied theory. Bayes' theorem is derived in the first three pages of the first chapter. Everything else in the book is aimed at helping the reader learn what it *means*. You won't even see a lot of the hairiest statistical code. Downey provides a set of libraries that implement the tool set of statistical analysis: distributions and their characteristics. The code in the book illustrates how to use those libraries to model and then solve the problem at hand.

Downey displays a sense of lightness and humor in his selection of many of the problems and his approach to the solutions (though the Kidney Tumor problem was rather more somber). The problems include calculating the best solution to the Monty Hall problem, finding where a hidden paint-ball opponent is located using scatter of the paint ball hits on the walls of an arena, and estimating the number of bacterial species that inhabit the human belly button, the last from a real survey of human microfauna. If nothing else, the set of questions he addresses will provide hours of fun for curious geeks like me.

The real lesson in *Think Bayes* is how to recognize problems that are suited to Bayesian analysis and then how to model them. Building the model in code leads to a computable solution. This makes it relatively easy to understand the characteristics of the problem by tweaking the model or the inputs and observing how that affects the output. Downey uses the notation of continuous math when it is useful to describe a problem, but he concentrates on discrete solutions that are susceptible to computational solution. In the end, the reader (and experimenter) will come away with a deep practical understanding of this increasingly common set of analytical tools.

Becoming Functional

Joshua Backfield
O'Reilly Media, Inc., 2014; 135 pages
ISBN-13 978-144936817-3

Reviewed by Mark Lamourine

The proponents of functional programming have been gaining strength in recent years. Pure functional languages like Haskell are being used in production environments. Erlang, while not 100% functional, has strong functional traits and is heavily used in the telco industry. Functional features are being added to existing imperative languages such as Java, and these are giving the champions of functionality more room to play. Even Scheme and Lisp, the most venerable of functional languages, have had an academic niche for decades but are finding wider use.

Backfield isn't trying to claim you should use a pure functional language like Scheme or Haskell, or even adopt strict functional style in all cases. Rather, his goal is to demonstrate the tenets of functional programming using a mixture of imperative languages with some functional features (Java 7) as well as a couple of more functional languages (Groovy and Scala). Java 8 is getting full functional features like lambdas and closures, but Backfield avoids giving more than one or two examples in Java 8 because the Java 7 user base is well established and will have a long life even after 8 is released. His method is to introduce each concept in the context of refactoring some existing imperative code. This is in fairly stark contrast to some other books that teach functional programming using only formal lambda calculus and a pure functional language.

In the first chapter Backfield introduces the major techniques of functional programming. In each of the following chapters he details these techniques and contrasts them to the equivalent imperative code to do the same job. He also presents a chapter called "Functional OOP," showing how objects can still be used to contain related data while using class methods to provide namespacing for the related functions. In the final chapter, he offers an outline of a refactoring plan, first recognizing the imperative patterns and then applying the appropriate functional transformation.

The book is pretty slim for the depth of the content. Backfield doesn't spend any time on language syntax or constructs except as he applies them to the example at hand. This book probably is not a good choice for a beginning coder. Someone with multiple-language experience shouldn't have any problem though. I'm familiar with Java but not with either Groovy or Scala. The syntax is clear enough that this did not get in the way of understanding the point of each example.

I must say I'm not yet sold on the idea that functional programming is universally superior to traditional imperative style. Clearly, each has value. People don't naturally think in a functional style. It takes significant training and practice to do it well. Functional programming techniques like statement chaining quickly become clever obscurities unless they are well commented.

That said, *Becoming Functional* provides a good introduction to functional programming technique without going too deeply into theory. It's a book that I will probably keep nearby to help me recognize and exploit opportunities to use functional programming constructs where they seem to be the best solution to a problem.

A Go Developer's Notebook

Eleanor McHugh
Lean Publishing, 2014; 84 pages (and counting)
https://leanpub.com/GoNotebook

Reviewed by Mark Lamourine

Some interesting things are happening in the publishing world. Publishers large and small are experimenting with alternate ways of writing and distributing books. One recent trend is the release of "rough cuts" or "beta versions" of technical manuals. The maturation of the ebook and e-readers has made this possible and even easy. Some publishers have found that offering early access to new texts and inviting comment both drums up interest and improves the final result. There is even a new breed of publishers that uses this public development model as their core business. Lean Publishing is one of these, and Lean is where Eleanor McHugh is writing *A Go Developer's Notebook*.

I first encountered *A Go Developer's Notebook* in an announcement in the Go+ community on Google Plus. The book has its own community now as well, where readers make comments and Eleanor posts updates and progress reports.

McHugh starts off typically with the Go version of "Hello World," but she dwells on it as something more than a cliché. Through the first chapter, she enhances the simple CLI program until it's a small Web server that can respond with a customized hello based on the queries it receives. It can serve both HTTP and HTTPS running in concurrent routines, and it includes a signal handler to shut down the services cleanly when the process is interrupted. This is rather a lot to pack into an introductory chapter. The second chapter, entitled "Echo," is just as packed, covering CLI and environment input and string management.

The writing style and progression of examples are engaging and interesting. They don't always follow a traditional sequence, but they are coherent and introduce useful concepts. They also serve as an introductory survey of commonly useful standard packages and modules.

Part way through Chapter 2 is where the nature of the writing and publishing process becomes evident. This really is McHugh's notebook. She's clearly got an outline, but the chapter kind of peters out. The next chapter on Types picks up strong again. You're seeing the mind of the writer at work. Several of the chapters just have heading skeletons while others have sparse content.

There are only two more chapters that have significant content. The first provides examples of looping constructs in Go. The other is entitled "Software Machines" and seems to be about techniques of using goroutines to create simple machines like stacks, queues, and processor simulations. Neither one contains any of the normal explanatory texts yet. The code provides examples of more complex behaviors and usage. It takes some work to understand what it is meant to do, but they are definitely interesting to read.

A Go Developer's Notebook wouldn't be a bad introduction to Go syntax for an experienced coder. It's not nearly a complete text but what is there promises to become something good.

In a previous decade, this review would have been an indictment, not a recommendation, but then you would have only gotten one copy in paper with no possibility of getting updates or giving feedback. The author would have had to do a lot of up-front writing or pitch an idea to a publisher before getting any sense of whether readers would be interested.

With a service like Leanpub, the authors can put incomplete but promising ideas and the text in front of readers directly. They can "test" text and get responses from readers. They can do incremental updates to approach a working document.

This is the way software development works. Writing for humans too, but until recently it was always hidden behind editors and publishers. In traditional publishing, it would be unacceptable to let the reader pay for something that was flawed and incomplete.

The way Leanpub works, the author registers and creates the template for her book, sets the title, and uploads the content in whatever state it is in. She sets pricing and can also offer a preview chapter. When someone purchases a book, he gets a copy in the current state. He also gets email notifications of updates as they come. The author gets 90% of the payment and retains all of her rights. She is free to take the text to a traditional publisher at any time.

The pricing model is also flexible. The author sets a minimum price but can also suggest a retail price. McHugh has posted a minimum price of $6 US and a retail request of $22 US. The buyer decides how much the book (and updates) are worth. The author is paid a "royalty" of 90% after a base transaction fee of $0.50 US on the actual amount paid. That is, on each transaction, Leanpub keeps $0.50 US plus 10% of the remainder. This ensures that Leanpub gets something substantial from each transaction and encourages the authors to set a reasonable minimum price, low enough to feel reasonable to the buyer, but high enough to see some return for each sale.

Leanpub and other ebook self-publishing sites offer a good place for budding authors to float ideas and practice their writing. They also look like a good place to fish for new and different takes on all kinds of subjects, as long as you're aware of what you're looking at: the seeds, not the trees.