Hands-On System Programming with Linux

Explore Linux system programming interfaces, theory, and practice

Packt
www.packt.com

Kaiwan N Billimoria

Hands-On System Programming with Linux

Explore Linux system programming interfaces, theory, and practice

Kaiwan N Billimoria

BIRMINGHAM - MUMBAI

Hands-On System Programming with Linux

Commissioning Editor: Gebin George
Acquisition Editor: Rohit Rajkumar
Content Development Editor: Priyanka Deshpande
Technical Editor: Rutuja Patade
Copy Editor: Safis Editing
Project Coordinator: Drashti Panchal
Proofreader: Safis Editing
Indexer: Rekha Nair
Graphics: Tom Scaria
Production Coordinator: Arvindkumar Gupta

First published: October 2018

Production reference: 1311018

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-78899-847-5

www.packtpub.com

mapt.io

Mapt is an online digital library that gives you full access to over 5,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Mapt is fully searchable
- Copy and paste, print, and bookmark content

Packt.com

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.packt.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Contributors

About the author

Kaiwan N Billimoria taught himself programming on his dad's IBM PC back in 1983. He was programming in C and Assembly on DOS until he discovered the joys of Unix (via Richard Steven's iconic book, *UNIX Network Programming*, and by writing C code on SCO Unix).

Kaiwan has worked on many aspects of the Linux system programming stack, including Bash scripting, system programming in C, kernel internals, and embedded Linux work. He has actively worked on several commercial/OSS projects. His contributions include drivers to the mainline Linux OS, and many smaller projects hosted on GitHub. His Linux passion feeds well into his passion for teaching these topics to engineers, which he has done for over two decades now. It doesn't hurt that he is a recreational ultra-marathoner too.

Writing a book is a lot of hard work, tightly coupled with teamwork. My deep gratitude to the team at Packt: Rohit, Priyanka, and Rutuja, as well as the technical reviewer, Tigran, and so many other behind-the-scenes workers. Of course, none of this would have been remotely possible without support from my family: my parents, Diana and Nadir; my brother, Darius; my wife, Dilshad; and my super kids, Sheroy and Danesh! Heartfelt thanks to you all.

About the reviewer

Tigran Aivazian has a master's degree in computer science and a master's degree in theoretical physics. He has written BFS and Intel microcode update drivers that have become part of the official Linux kernel. He is the author of a book titled *Linux 2.4 Kernel Internals*, which is available in several languages on the Linux documentation project. He worked at Veritas as a Linux kernel architect, improving the kernel and teaching OS internals. Besides technological pursuits, Tigran has produced scholarly Bible editions in Hebrew, Greek, Syriac, Slavonic, and ancient Armenian. Recently, he published *The British Study Edition of the Urantia Papers*. He is currently working on the foundations of quantum mechanics in a branch of physics called quantum infodynamics.

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packtpub.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Table of Contents

Preface

The Linux OS and its embedded and server applications are critical components of today's key software infrastructure in a decentralized and networked universe. Industry demand for proficient Linux developers is ever-increasing. This book aims to give you two things: a solid theoretical base, and practical, industry-relevant information—illustrated by code—covering the Linux system programming domain. This book delves into the art and science of Linux system programming, including system architecture, virtual memory, process memory and management, signaling, timers, multithreading, scheduling, and file I/O.

This book attempts to go beyond the use API X to do Y approach; it takes pains to explain the concepts and theory required to understand the programming interfaces, the design decisions, and trade-offs made by experienced developers when using them and the rationale behind them. Troubleshooting tips and industry best practices round out the book's coverage. By the end of this book, you will have the conceptual knowledge, as well as the hands-on experience, needed for working with Linux system programming interfaces.

Who this book is for

Hands-On System Programming with Linux is for Linux professionals: system engineers, programmers, and testers (QA). It's also for students; anyone, really, who wants to go beyond using an API set to understand the theoretical underpinnings and concepts behind the powerful Linux system programming APIs. You should be familiar with Linux at the user level, including aspects such as logging in, using the shell via the command-line interface, and using tools such as find, grep, and sort. A working knowledge of the C programming language is required. No prior experience with Linux systems programming is assumed.

What this book covers

`Chapter 1`, *Linux System Architecture*, covers the key basics: the Unix design philosophy and the Linux system architecture. Along the way, other important aspects—CPU privilege levels, the processor ABI, and what system calls really are—are dealt with.

`Chapter 2`, *Virtual Memory*, dives into clearing up common misconceptions about what virtual memory really is and why it is key to modern OS design; the layout of the process virtual address space is covered too.

`Chapter 3`, *Resource Limits*, delves into the topic of per-process resource limits and the APIs governing their usage.

`Chapter 4`, *Dynamic Memory Allocation*, initially covers the basics of the popular malloc family of APIs, then dives into more advanced aspects, such as the program break, how malloc really behaves, demand paging, memory locking and protection, and using the alloca function.

`Chapter 5`, *Linux Memory Issues*, introduces you to the (unfortunately) prevalent memory defects that end up in our projects due to a lack of understanding of the correct design and use of memory APIs. Defects such as undefined behavior (in general), overflow and underflow bugs, leakage, and others are covered.

`Chapter 6`, *Debugging Tools for Memory Issues*, shows how to leverage existing tools, including the compiler itself, Valgrind, and AddressSanitizer, which is used to detect the memory issues you will have seen in the previous chapter.

`Chapter 7`, *Process Credentials*, is the first of two chapters focused on having you think about and understand security and privilege from a system perspective. Here, you'll learn about the traditional security model – a set of process credentials – as well as the APIs for manipulating them. Importantly, the concepts of setuid-root processes and their security repercussions are delved into.

`Chapter 8`, *Process Capabilities*, introduces you to the modern POSIX capabilities model and how security can benefit when application developers learn to use and leverage this model instead of the traditional model (seen in the previous chapter). What capabilities are, how to embed them, and practical design for security is also looked into.

`Chapter 9`, *Process Execution*, is the first of four chapters dealing with the broad area of process management (execution, creation, and signaling). In this particular chapter, you'll learn how the (rather unusual) Unix exec axiom behaves and how to use the API set (the exec family) to exploit it.

`Chapter 10`, *Process Creation,* delves into how exactly the `fork(2)` system call behaves and should be used; we depict this via our seven rules of fork. The Unix fork-exec-wait semantic is described (diving into the wait APIs as well), orphan and zombie processes are also covered.

`Chapter 11`, *Signaling – Part I,* deals with the important topic of signals on the Linux platform: the what, the why, and the how. We cover the powerful `sigaction(2)` system call here, along with topics such as reentrant and signal-async safety, sigaction flags, signal stacks, and others.

`Chapter 12`, *Signaling – Part II,* continues our coverage of signaling, what with it being a large topic. We take you through the correct way to write a signal handler for the well-known and fatal segfault, working with real-time signals, delivering signal to processes, performing IPC with signals, and alternate means to handle signals.

`Chapter 13`, *Timers,* teaches you about the important (and signal-related) topic of how to set up and handle timers in real-world Linux applications. We first cover the traditional timer APIs and quickly move onto the modern POSIX interval timers and how to use them to this end. Two interesting, small projects are presented and walked through.

`Chapter 14`, *Multithreading with Pthreads Part I – Essentials,* is the first of a trilogy on multithreading with the pthreads framework on Linux. Here, we introduce you to what exactly a thread is, how it differs from a process, and the motivation (in terms of design and performance) for using threads. The chapter then guides you through the essentials of writing a pthreads application on Linux ,covering thread creation, termination, joining, and more.

`Chapter 15`, *Multithreading with Pthreads Part II – Synchronization,* is a chapter dedicated to the really important topic of synchronization and race prevention. You will first understand the issue at hand, then delve into the key topics of atomicity, locking, deadlock prevention, and others. Next, the chapter teaches you how to use pthreads synchronization APIs with respect to the mutex lock and condition variables.

`Chapter 16`, *Multithreading with Pthreads Part III,* completes our work on multithreading; we shed light on the key topics of thread safety, thread cancellation and cleanup, and handling signals in a multithreaded app. We round off the chapter with a discussion on the pros and cons of multithreading and address some FAQs.

`Chapter 17`, *CPU Scheduling on Linux*, introduces you to scheduling-related topics that the system programmer should be aware of. We cover the Linux process/thread state machine, the notion of real time and the three (minimal) POSIX CPU scheduling policies that the Linux OS brings to the table. Exploiting the available APIs, you'll learn how to write a soft real-time app on Linux. We finish the chapter with a brief look at the (interesting!) fact that Linux *can* be patched to work as an RTOS.

`Chapter 18`, *Advanced File I/O*, is completely focused on the more advanced ways of performing IO on Linux in order to gain maximum performance (as IO is often the bottleneck). You are briefly shown how the Linux IO stack is architected (the page cache being critical), and the APIs that give advice to the OS on file access patterns. Writing IO code for performance, as you'll learn, involves the use of technologies such as SG-I/O, memory mapping, DIO, and AIO.

`Chapter 19`, *Troubleshooting and Best Practices*, is a critical summation of the key points to do with troubleshooting on Linux. You'll be briefed upon the use of powerful tools, such as perf and tracing tools. Then, very importantly, the chapter attempts to summarize key points on software engineering in general and programming on Linux in particular, looking at industry best practices. We feel these are critical takeaways for any programmer.

`Appendix A`, File I/O Essentials, introduces you to performing efficient file I/O on the Linux platform, via both the streaming (stdio library layer) API set as well as the underlying system calls. Along the way, important information on buffering and its effects on performance are covered.

For this chapter refer to: `https://www.packtpub.com/sites/default/files/downloads/File_IO_Essentials.pdf`.

`Appendix B`, Daemon Processes, introduces you, in a succinct fashion, to the world of the daemon process on Linux. You'll be shown how to write a traditional SysV-style daemon process. There is also a brief note on what is involved in constructing a modern, new-style daemon process.

For this chapter refer to: `https://www.packtpub.com/sites/default/files/downloads/Daemon_Processes.pdf`.

To get the most out of this book

As mentioned earlier, this book is targeted at both Linux software professionals—be they developers, programmers, architects, or QA staff members—as well as serious students looking to expand their knowledge and skills with the key topics of system programming on the Linux OS.

We assume that you are familiar with using a Linux system via the command-line interface, the shell. We also assume that you are familiar with programming in the C language, know how to use the editor and the compiler, and are familiar with the basics of the Makefile. We do *not* assume that you have any prior knowledge of the topics covered in the book.

To get the most out of this book—and we are very clear on this point—you must not just read the material, but must also actively work on, try out, and modify the code examples provided, and try and finish the assignments as well! Why?
Simple: doing is what really teaches you and internalizes a topic; making mistakes and fixing them being an essential part of the learning process. We always advocate an empirical approach—don't take anything at face value. Experiment, try it out for yourself, and see.

To this end, we urge you to clone this book's GitHub repository (see the following section for instructions), browse through the files, and try them out. Using a **Virtual Machine** (**VM**) for experimentation is (quite obviously) definitely recommended (we have tested the code on both Ubuntu 18.04 LTS and Fedora 27/28). A listing of mandatory and optional software packages to install on the system is also provided within the book's GitHub repository; please read through and install all required utilities to get the best experience.

Last, but definitely not least, each chapter has a *Further reading* section, where additional online links and books (in some cases) are mentioned; we urge you to browse through these. You will find the *Further reading* material for each chapter available on the book's GitHub repository.

Download the example code files

You can download the example code files for this book from your account at `www.packt.com`. If you purchased this book elsewhere, you can visit `www.packt.com/support` and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at `www.packt.com`.
2. Select the **SUPPORT** tab.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/Hands-on-System-Programming-with-Linux`. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out.

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: `https://www.packtpub.com/sites/default/files/downloads/9781788998475_ColorImages.pdf`

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "Let's check these out via the source code of our `membugs.c` program."

A block of code is set as follows:

```
include <pthread.h>
int pthread_mutexattr_gettype(const pthread_mutexattr_t *restrict
attr,     int *restrict type);
int pthread_mutexattr_settype(pthread_mutexattr_t *attr, int type);
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
include <pthread.h>
int pthread_mutexattr_gettype(const pthread_mutexattr_t *restrict
attr,      int *restrict type);
int pthread_mutexattr_settype(pthread_mutexattr_t *attr, int type);
```

Any command-line input or output is written as follows:

```
$ ./membugs 3
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "Select **C** as the language via the drop-down."

Warnings or important notes appear like this.

Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: Email `customercare@packtpub.com` and mention the book title in the subject of your message. If you have questions about any aspect of this book, please email us at `customercare@packtpub.com`.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit `www.packt.com/submit-errata`, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packt.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `authors.packtpub.com`.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit `packt.com`.

1
Linux System Architecture

This chapter informs the reader about the system architecture of the Linux ecosystem. It first conveys the elegant Unix philosophy and design fundamentals, then delves into the details of the Linux system architecture. The importance of the ABI, CPU privilege levels, and how modern **operating systems** (**OSes**) exploit them, along with the Linux system architecture's layering, and how Linux is a monolithic architecture, will be covered. The (simplified) flow of a system call API, as well as kernel-code execution contexts, are key points.

In this chapter, the reader will be taken through the following topics:

- The Unix philosophy in a nutshell
- Architecture preliminaries
- Linux architecture layers
- Linux—a monolithic OS
- Kernel execution contexts

Along the way, we'll use simple examples to make the key philosophical and architectural points clear.

Technical requirements

A modern desktop PC or laptop is required; Ubuntu Desktop specifies the following as recommended system requirements for installation and usage of the distribution:

- 2 GHz dual core processor or better
- RAM
    - **Running on a physical host**: 2 GB or more system memory
    - **Running as a guest**: The host system should have at least 4 GB RAM (the more, the better and smoother the experience)

- 25 GB of free hard drive space
- Either a DVD drive or a USB port for the installer media
- Internet access is definitely helpful

We recommend the reader use one of the following Linux distributions (can be installed as a guest OS on a Windows or Linux host system, as mentioned):

- Ubuntu 18.04 LTS Desktop (Ubuntu 16.04 LTS Desktop is a good choice too as it has long term support as well, and pretty much everything should work)
    - Ubuntu Desktop download link: `https://www.ubuntu.com/download/desktop`
- Fedora 27 (Workstation)
    - Download link: `https://getfedora.org/en_GB/workstation/download/`

Note that these distributions are, in their default form, OSS and non-proprietary, and free to use as an end user.

There are instances where the entire code snippet isn't included in the book . Thus the GitHub URL to refer the codes: `https://github.com/PacktPublishing/Hands-on-System-Programming-with-Linux`.
Also, for the *Further reading* section, refer to the preceding GitHub link.

Linux and the Unix operating system

Moore's law famously states that the number of transistors in an IC will double (approximately) every two years (with an addendum that the cost would halve at pretty much the same rate). This law, which remained quite accurate for many years, is one of the things that clearly underscored what people came to realize, and even celebrate, about the electronics and the **Information Technology** (**IT**) industry; the sheer speed with which innovation and paradigm shifts in technology occur here is unparalleled. So much so that we now hardly raise an eyebrow when, every year, even every few months in some cases, new innovations and technology appear, challenge, and ultimately discard the old with little ceremony.

Against this backdrop of rapid all-consuming change, there lives an engaging anomaly: an OS whose essential design, philosophy, and architecture have changed hardly at all in close to five decades. Yes, we are referring to the venerable Unix operating system.

Organically emerging from a doomed project at AT&T's Bell Labs (Multics) in around 1969, Unix took the world by storm. Well, for a while at least.

But, you say, this is a book about Linux; why all this information about Unix? Simply because, at heart, Linux is the latest avatar of the venerable Unix OS. Linux is a Unix-like operating system (among several others). The code, by legal necessity, is unique; however, the design, philosophy, and architecture of Linux are pretty much identical to those of Unix.

The Unix philosophy in a nutshell

To understand anyone (or anything), one must strive to first understand their (or its) underlying philosophy; to begin to understand Linux is to begin to understand the Unix philosophy. Here, we shall not attempt to delve into every minute detail; rather, an overall understanding of the essentials of the Unix philosophy is our goal. Also, when we use the term Unix, we very much also mean Linux!

The way that software (particularly, tools) is designed, built, and maintained on Unix slowly evolved into what might even be called a pattern that stuck: the Unix design philosophy. At its heart, here are the pillars of the Unix philosophy, design, and architecture:

- Everything is a process; if it's not a process, it's a file
- One tool to do one task
- Three standard I/O channel
- Combine tools seamlessly
- Plain text preferred
- CLI, not GUI
- Modular, designed to be repurposed by others
- Provide the mechanism, not the policy

Let's examine these pillars a little more closely, shall we?

Everything is a process – if it's not a process, it's a file

A process is an instance of a program in execution. A file is an object on the filesystem; beside regular file with plain text or binary content; it could also be a directory, a symbolic link, a device-special file, a named pipe, or a (Unix-domain) socket.

The Unix design philosophy abstracts peripheral devices (such as the keyboard, monitor, mouse, a sensor, and touchscreen) as files – what it calls device files. By doing this, Unix allows the application programmer to conveniently ignore the details and just treat (peripheral) devices as though they are ordinary disk files.

The kernel provides a layer to handle this very abstraction – it's called the **Virtual Filesystem Switch** (**VFS**). So, with this in place, the application developer can open a device file and perform I/O (reads and writes) upon it, all using the usual API interfaces provided (relax, these APIs will be covered in a subsequent chapter).

In fact, every process inherits three files on creation:

- **Standard input** (`stdin`**: fd 0**): The keyboard device, by default
- **Standard output** (`stdout`**: fd 1**): The monitor (or terminal) device, by default
- **Standard error** (`stderr`**: fd 2**): The monitor (or terminal) device, by default

fd is the common abbreviation, especially in code, for **file descriptor**; it's an integer value that refers to the open file in question.

Also, note that we mention it's a certain device by default – this implies the defaults can be changed. Indeed, this is a key part of the design: changing standard input, output, or error channels is called **redirection**, and by using the familiar <, > and 2> shell operators, these file channels are redirected to other files or devices.

On Unix, there exists a class of programs called **filters**.

A filter is a program that reads from its standard input, possibly modifies the input, and writes the filtered result to its standard output.

Filters on Unix are very common utilities, such as `cat`, `wc`, `sort`, `grep`, `perl`, `head`, and `tail`.

Filters allow Unix to easily sidestep design and code complexity. How?

Let's take the `sort` filter as a quick example. Okay, we'll need some data to sort. Let's say we run the following commands:

```
$ cat fruit.txt
orange
banana
apple
pear
grape
pineapple
lemon
cherry
papaya
mango
$
```

Now we consider four scenarios of using `sort`; based on the parameter(s) we pass, we are actually performing explicit or implicit input-, output-, and/or error-redirection!

Scenario 1: Sort a file alphabetically (one parameter, input implicitly redirected to file):

```
$ sort fruit.txt
 apple
 banana
 cherry
 grape
 lemon
 mango
 orange
 papaya
 pear
 pineapple
$
```

All right!

Hang on a second, though. If `sort` is a filter (and it is), it should read from its `stdin` (the keyboard) and write to its `stdout` (the terminal). It is indeed writing to the terminal device, but it's reading from a file, `fruit.txt`.

This is deliberate; if a parameter is provided, the sort program treats it as standard input, as clearly seen.

Also, note that `sort fruit.txt` is identical to `sort < fruit.txt`.

Scenario 2: Sort any given input alphabetically (no parameters, input and output from and to stdin/stdout):

```
$ sort
mango
apple
pear
^D
apple
mango
pear
$
```

Once you type `sort` and press the *Enter* key, and the sort process comes alive and just waits. Why? It's waiting for you, the user, to type something. Why? Recall, every process by default reads its input from standard input or stdin – the keyboard device! So, we type in some fruit names. When we're done, press *Ctrl + D*. This is the default character sequence that signifies **end-of-file** (**EOF**), or in cases such as this, end-of-input. Voila! The input is sorted and written. To where? To the `sort` process's stdout – the terminal device, hence we see it.

Scenario 3: Sort any given input alphabetically and save the output to a file (explicit output redirection):

```
$ sort > sorted.fruit.txt
mango
apple
pear
^D
$
```

Similar to Scenario 2, we type in some fruit names and then *Ctrl + D* to tell sort we're done. This time, though, note that the output is redirected (via the > meta-character) to the `sorted.fruits.txt` file!

So, as expected is the following output:

```
$ cat sorted.fruit.txt
apple
mango
pear
$
```

Scenario 4: Sort a file alphabetically and save the output and errors to a file (explicit input-, output-, and error-redirection):

```
$ sort < fruit.txt > sorted.fruit.txt 2> /dev/null
$
```

Interestingly, the end result is the same as in the preceding scenario, with the added advantage of redirecting any error output to the error channel. Here, we redirect the error output (recall that file descriptor 2 always refers to `stderr`) to the `/dev/null` special device file; `/dev/null` is a device file whose job is to act as a sink (a black hole). Anything written to the null device just disappears forever! (Who said there isn't magic on Unix?) Also, its complement is `/dev/zero`; the zero device is a source – an infinite source of zeros. Reading from it returns zeroes (the first ASCII character, not numeric 0); it has no end-of-file!

One tool to do one task

In the Unix design, one tries to avoid creating a Swiss Army knife; instead, one creates a tool for a very specific, designated purpose and for that one purpose only. No ifs, no buts; no cruft, no clutter. This is design simplicity at its best.

> *"Simplicity is the ultimate sophistication."*
>
> *- Leonardo da Vinci*

Take a common example: when working on the Linux **CLI** (**command-line interface**), you would like to figure out which of your locally mounted filesystems has the most available (disk) space.

We can get the list of locally mounted filesystems by an appropriate switch (just `df` would do as well):

```
$ df --local
Filesystem                1K-blocks    Used Available Use% Mounted on
rootfs                     20640636 1155492  18436728   6% /
udev                          10240       0     10240   0% /dev
tmpfs                         51444     160     51284   1% /run
tmpfs                          5120       0      5120   0% /run/lock
tmpfs                        102880       0    102880   0% /run/shm
$
```

To sort the output, one would need to first save it to a file; one could use a temporary file for this purpose, tmp, and then sort it, using the `sort` utility, of course. Finally, we delete the offending temporary file. (Yes, there's a better way, piping; refer to the, *Combine tools seamlessly* section)

Note that the available space is the fourth column, so we sort accordingly:

```
$ df --local > tmp
$ sort -k4nr tmp
rootfs                     20640636 1155484  18436736   6% /
tmpfs                        102880       0    102880   0% /run/shm
tmpfs                         51444     160     51284   1% /run
udev                          10240       0     10240   0% /dev
tmpfs                          5120       0      5120   0% /run/lock
Filesystem                1K-blocks    Used Available Use% Mounted on
$
```

Whoops! The output includes the heading line. Let's first use the versatile `sed` utility – a powerful non-interactive editor tool – to eliminate the first line, the header, from the output of `df`:

```
$ df --local > tmp
$ sed --in-place '1d' tmp
$ sort -k4nr tmp
rootfs                     20640636 1155484  18436736   6% /
tmpfs                        102880       0    102880   0% /run/shm
tmpfs                         51444     160     51284   1% /run
udev                          10240       0     10240   0% /dev
tmpfs                          5120       0      5120   0% /run/lock
$ rm -f tmp
```

So what? The point is, on Unix, there is no one utility to list mounted filesystems and sort them by available space simultaneously.

Instead, there is a utility to list mounted filesystems: `df`. It does a great job of it, with option switches to choose from. (How does one know which options? Learn to use the man pages, they're extremely useful.)

There is a utility to sort text: `sort`. Again, it's the last word in sorting text, with plenty of option switches to choose from for pretty much every conceivable sort one might require.

The Linux man pages: **man** is short for **manual**; on a Terminal window, type `man man` to get help on using man. Notice the manual is divided into 9 sections. For example, to get the manual page on the stat system call, type `man 2 stat` as all system calls are in section 2 of the manual. The convention used is cmd or API; thus, we refer to it as `stat(2)`.

As expected, we obtain the results. So what exactly is the point? It's this: we used three utilities, not one. `df` , to list the mounted filesystems (and their related metadata), `sed`, to eliminate the header line, and `sort`, to sort whatever input its given (in any conceivable manner).

`df` can query and list mounted filesystems, but it cannot sort them. `sort` can sort text; it cannot list mounted filesystems.

Think about that for a moment.

Combine them all, and you get more than the sum of its parts! Unix tools typically do one task and they do it to its logical conclusion; no one does it better!

Having said this, I would like to point out – a tiny bit sheepishly – the highly renowned tool Busybox. Busybox (`http://busybox.net`) is billed as The Swiss Army Knife of Embedded Linux. It is indeed a very versatile tool; it has its place in the embedded Linux ecosystem – precisely because it would be too expensive on an embedded box to have separate binary executables for each and every utility (and it would consume more RAM). Busybox solves this problem by having a single binary executable (along with symbolic links to it from each of its applets, such as ls, ps, df, and sort).
So, nevertheless, besides the embedded scenario and all the resource limitations it implies, do follow the *One tool to do one task* rule!

Three standard I/O channels

Several popular Unix tools (technically, filters) are, again, deliberately designed to read their input from a standard file descriptor called **standard input** (**stdin**) – possibly modify it, and write their resultant output to a standard file descriptor **standard output** (**stdout**). Any error output can be written to a separate error channel called **standard error** (**stderr**).

In conjunction with the shell's redirection operators (`>` for output-redirection and `<` for input-redirection, `2>` for stderr redirection), and even more importantly with piping (refer section, *Combine tools seamlessly*), this enables a program designer to highly simplify. There's no need to hardcode (or even softcode, for that matter) input and output sources or sinks. It just works, as expected.

Let's review a couple of quick examples to illustrate this important point.

Word count

How many lines of source code are there in the C `netcat.c` source file I downloaded? (Here, we use a small part of the popular open source `netcat` utility code base.) We use the `wc` utility. Before we go further, what's wc? **word count** (**wc**) is a filter: it reads input from stdin, counts the number of lines, words, and characters in the input stream, and writes this result to its stdout. Further, as a convenience, one can pass filenames as parameters to it; passing the `-l` option switch has wc only print the number of lines:

```
$ wc -l src/netcat.c
618 src/netcat.c
$
```

Here, the input is a filename passed as a parameter to `wc`.

Interestingly, we should by now realize that if we do not pass it any parameters, `wc` would read its input from stdin, which by default is the keyboard device. For example is shown as follows:

```
$ wc -l
hey, a small
quick test
 of reading from stdin
by wc!
^D
4
$
```

Yes, we typed in `4` lines to stdin; thus the result is 4, written to stdout – the terminal device by default.

Here is the beauty of it:

```
$ wc -l < src/netcat.c > num
$ cat num
618
$
```

As we can see, wc is a great example of a Unix filter.

cat

Unix, and of course Linux, users learn to quickly get familiar with the daily-use `cat` utility. At first glance, all cat does is spit out the contents of a file to the terminal.

For example, say we have two plain text files, `myfile1.txt` and `myfile2.txt`:

```
$ cat myfile1.txt
Hello,
Linux System Programming,
World.
$ cat myfile2.txt
Okey dokey,
bye now.
$
```

Okay. Now check this out:

```
$ cat myfile1.txt myfile2.txt
Hello,
Linux System Programming,
```

```
World.
Okey dokey,
bye now.
$
```

Instead of needing to run `cat` twice, we ran it just once, by passing the two filenames to it as parameters.

In theory, one can pass any number of parameters to cat: it will use them all, one by one!

Not just that, one can use shell wildcards too (`*` and `?`; in reality, the shell will first expand the wildcards, and pass on the resultant path names to the program being invoked as parameters):

```
$ cat myfile?.txt
Hello,
Linux System Programming,
World.
Okey dokey,
bye now.
$
```

This, in fact, illustrates another key point: any number of parameters or none is considered the right way to design a program. Of course, there are exceptions to every rule: some programs demand mandatory parameters.

Wait, there's more. `cat` too, is an excellent example of a Unix filter (recall: a filter is a program that reads from its standard input, modifies its input in some manner, and writes the result to its standard output).

So, quick quiz, if we just run `cat` with no parameters, what would happen?
Well, let's try it out and see:

```
$ cat
hello,
hello,
oh cool
oh cool
it reads from stdin,
it reads from stdin,
and echoes whatever it reads to stdout!
and echoes whatever it reads to stdout!
ok bye
ok bye
^D
$
```

Wow, look at that: `cat` blocks (waits) at its stdin, the user types in a string and presses the Enter key, `cat` responds by copying its stdin to its stdout – no surprise there, as that's the job of cat in a nutshell!

One realizes the commands shown as follows:

- `cat fname` is the same as `cat < fname`
- `cat > fname` creates or overwrites the `fname` file

There's no reason we can't use cat to append several files together:

```
$ cat fname1 fname2 fname3 > final_fname
$
```

There's no reason this must be done with only plain text files; one can join together binary files too.

In fact, that's what the utility does – it concatenates files. Thus its name; as is the norm on Unix, is highly abbreviated – from concatenate to just cat. Again, clean and elegant – the Unix way.

cat shunts out file contents to stdout, in order. What if one wants to display a file's contents in reverse order (last line first)? Use the Unix `tac` utility – yes, that's cat spelled backward!

Also, FYI, we saw that cat can be used to efficiently join files. Guess what: the `split (1)` utility can be used to break a file up into pieces.

Combine tools seamlessly

We just saw that common Unix utilities are often designed as filters, giving them the ability to read from their standard input and write to their standard output. This concept is elegantly extended to seamlessly combine together multiple utilities, using an IPC mechanism called a **pipe**.

Also, we recall that the Unix philosophy embraces the do one task only design. What if we have one program that does task A and another that does task B and we want to combine them? Ah, that's exactly what pipes do! Refer to the following code:

```
prg_does_taskA | prg_does_taskB
```

A pipe essentially is redirection performed twice: the output of the left-hand program becomes the input to the right-hand program. Of course, this implies that the program on the left must write to stdout, and the program on the read must read from stdin.

An example: sort the list of mounted filesystems by space available (in reverse order).

As we have already discussed this example in the *One tool to do one task* section, we shall not repeat the same information.

Option 1: Perform the following code using a temporary file (refer section, *One tool to do one task*):

```
$ df --local | sed '1d' > tmp
$ sed --in-place '1d' tmp
$ sort -k4nr tmp
rootfs 20640636 1155484 18436736 6% /
tmpfs 102880 0 102880 0% /run/shm
tmpfs 51444 160 51284 1% /run
udev 10240 0 10240 0% /dev
tmpfs 5120 0 5120 0% /run/lock
$ rm -f tmp
```

Option 2 : Using pipes—clean and elegant:

```
$ df --local | sed '1d' | sort -k4nr
rootfs                    20640636 1155492  18436728   6% /
tmpfs                       102880       0    102880   0% /run/shm
tmpfs                        51444     160     51284   1% /run
udev                         10240       0     10240   0% /dev
tmpfs                         5120       0      5120   0% /run/lock
$
```

Not only is this elegant, it is also far superior performance-wise, as writing to memory (the pipe is a memory object) is much faster than writing to disk.

One can extend this notion and combine multiple tools over multiple pipes; in effect, one can build a super tool from several regular tools by combining them.

As an example: display the three processes taking the most (physical) memory; only display their PID, **virtual size** (**VSZ**), **resident set size** (**RSS**) (RSS is a fairly accurate measure of physical memory usage), and the name:

```
$ ps au | sed '1d' | awk '{printf("%6d %10d %10d %-32s\n", $2, $5, $6,
$11)}' | sort -k3n | tail -n3
 10746    3219556     665252 /usr/lib64/firefox/firefox
 10840    3444456    1105088 /usr/lib64/firefox/firefox
  1465    5119800    1354280 /usr/bin/gnome-shell
$
```

Here, we've combined five utilities, `ps`, `sed`, `awk`, `sort`, and `tail`, over four pipes. Nice!

Another example: display the process, not including daemons*, taking up the most memory (RSS):

```
ps aux | awk '{if ($7 != "?") print $0}' | sort -k6n | tail -n1
```

A daemon is a system background process; we'll cover this concept in *Daemon Process* here: `https://www.packtpub.com/sites/default/files/downloads/Daemon_Processes.pdf`.

Plain text preferred

Unix programs are generally designed to work with text as it's a universal interface. Of course, there are several utilities that do indeed operate on binary objects (such as object and executable files); we aren't referring to them here. The point is this: Unix programs are designed to work on text as it simplifies the design and architecture of the program.

A common example: an application, on startup, parses a configuration file. The configuration file could be formatted as a binary blob. On the other hand, having it as a plain text file renders it easily readable (invaluable!) and therefore easier to understand and maintain. One might argue that parsing binary would be faster. Perhaps to some extent this is so, but consider the following:

- With modern hardware, the difference is probably not significant
- A standardized plain text format (such as XML) would have optimized code to parse it, yielding both benefits

Remember, simplicity is key!

CLI, not GUI

The Unix OS, and all its applications, utilities, and tools, were always built to be used from a **command-line-interface** (**CLI**), typically, the shell. From the 1980s onward, the need for a **Graphical User Interface** (**GUI**) became apparent.

Robert Scheifler of MIT, considered the chief design architect behind the X Window System, built an exceedingly clean and elegant architecture, a key component of which is this: the GUI forms a layer (well, actually, several layers) above the OS, providing libraries for GUI clients, that is, applications.

The GUI was never designed to be intrinsic to applications or the OS—it's always optional.

This architecture still holds up today. Having said that, especially on embedded Linux, performance reasons are seeing the advent of newer architectures, such as the frame buffer and Wayland. Also, though Android, which uses the Linux kernel, necessitates a GUI for the end user, the system developer's interface to Android, ADB, is a CLI.

A huge number of production-embedded and server Linux systems run purely on CLI interfaces. The GUI is almost like an add-on feature, for the end user's ease of operation.

Wherever appropriate, design your tools to work in the CLI environment; adapting it into a GUI at a later point is then straightforward.
Cleanly and carefully separating the business logic of the project or product from its GUI is a key to good design.

Modular, designed to be repurposed by others

From its very early days, the Unix OS was deliberately designed and coded with the tacit assumption that multiple programmers would work on the system. Thus, the culture of writing clean, elegant, and understandable code, to be read and worked upon by other competent programmers, was ingrained.

Later, with the advent of the Unix wars, proprietary and legal concerns overrode this sharing model. Interestingly, history shows that the Unix's were fading in relevance and industry use, until the timely advent of none other than the Linux OS – an open source ecosystem at its very best! Today, the Linux OS is widely acknowledged as the most successful GNU project. Ironic indeed!

Provide mechanisms, not policies

Let's understand this principle with a simple example.

When designing an application, you need to have the user enter a login `name` and `password`. The function that performs the work of getting and checking the password is called, let's say, `mygetpass()`. It's invoked by the `mylogin()` function: `mylogin()` → `mygetpass()`.

Now, the protocol to be followed is this: if the user gets the password wrong three times in a row, the program should not allow access (and should log the case). Fine, but where do we check this?

The Unix philosophy: do not implement the logic, if the password is specified wrongly three times, abort in the `mygetpass()` function. Instead, just have `mygetpass()` return a Boolean (true when the password is right, false when the password is wrong), and have the `mylogin()` calling function implement whatever logic is required.

Pseudocode

The following is the wrong approach:

```
mygetpass()
{
    numtries=1

    <get the password>
    if (password-is-wrong) {
         numtries ++
         if (numtries >= 3) {
            <write and log failure message>
            <abort>
         }
 }
 <password correct, continue>
```

```
}
mylogin()
{
    mygetpass()
}
```

Now let's take a look at the right approach: the Unix way! Refer to the following code:

```
mygetpass()
{
   <get the password>

   if (password-is-wrong)
      return false;

   return true;
}
mylogin()
{
    maxtries = 3

    while (maxtries--) {
       if (mygetpass() == true)
           <move along, call other routines>
    }

    // If we're here, we've failed to provide the
    // correct password
    <write and log failure message>
    <abort>
}
```

The job of `mygetpass()` is to get a password from the user and check whether it's correct; it returns success or failure to the caller – that's it. That's the mechanism. It is not its job to decide what to do if the password is wrong – that's the policy, and left to the caller.

Now that we've covered the Unix philosophy in a nutshell, what are the important takeaways for you, the system developer on Linux?

Learning from, and following, the Unix philosophy when designing and implementing your applications on the Linux OS will provide a huge payoff. Your application will do the following:

- Be a natural fit on the system; this is very important
- Have greatly reduced complexity

- Have a modular design that is clean and elegant
- Be far more maintainable

Linux system architecture

In order to clearly understand the Linux system architecture, one needs to first understand a few important concepts: the processor **Application Binary Interface** (**ABI**), CPU privilege levels, and how these affect the code we write. Accordingly, and with a few code examples, we'll delve into these here, before diving into the details of the system architecture itself.

Preliminaries

If one is posed the question, "what is the CPU for?", the answer is pretty obvious: the CPU is the heart of the machine – it reads in, decodes, and executes machine instructions, working on memory and peripherals. It does this by incorporating various stages.

Very simplistically, in the Instruction Fetch stage, it reads in machine instructions (which we represent in various human-readable ways – in hexadecimal, assembly, and high-level languages) from memory (RAM) or CPU cache. Then, in the Instruction Decode phase, it proceeds to decipher the instruction. Along the way, it makes use of the control unit, its register set, ALU, and memory/peripheral interfaces.

The ABI

Let's imagine that we write a C program, and run it on the machine.

Well, hang on a second. C code cannot possibly be directly deciphered by the CPU; it must be converted into machine language. So, we understand that on modern systems we will have a toolchain installed – this includes the compiler, linker, library objects, and various other tools. We compile and link the C source code, converting it into an executable format that can be run on the system.

The processor **Instruction Set Architecture** (**ISA**) – documents the machine's instruction formats, the addressing schemes it supports, and its register model. In fact, CPU **Original Equipment Manufacturers** (**OEMs**) release a document that describes how the machine works; this document is generally called the ABI. The ABI describes more than just the ISA; it describes the machine instruction formats, the register set details, the calling convention, the linking semantics, and the executable file format, such as ELF. Try out a quick Google for x86 ABI – it should reveal interesting results.

The publisher makes the full source code for this book available on their website; we urge the reader to perform a quick Git clone on the following URL. Build and try it: `https://github.com/PacktPublishing/Hands-on-System-Programming-with-Linux`.

Let's try this out. First, we write a simple `Hello, World` type of C program:

```
$ cat hello.c
/*
 * hello.c
 *
 ***************************************************************
 * This program is part of the source code released for the book
 *  "Linux System Programming"
 *  (c) Kaiwan N Billimoria
 *  Packt Publishers
 *
 * From:
 *  Ch 1 : Linux System Architecture
 ***************************************************************
 * A quick 'Hello, World'-like program to demonstrate using
 * objdump to show the corresponding assembly and machine
 * language.
 */
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main(void)
{
    int a;

    printf("Hello, Linux System Programming, World!\n");
    a = 5;
    exit(0);
}
$
```

We build the application via the `Makefile`, with `make`. Ideally, the code must compile with no warnings:

```
$ gcc -Wall -Wextra hello.c -o hello
hello.c: In function ‘main':
hello.c:23:6: warning: variable ‘a' set but not used [-Wunused-but-set-
variable]
  int a;
      ^
$
```

Important! Do not ignore compiler warnings with production code. Strive to get rid of all warnings, even the seemingly trivial ones; this will help a great deal with correctness, stability, and security.

In this trivial example code, we understand and anticipate the unused variable warning that `gcc` emits, and just ignore it for the purpose of this demo.

The exact warning and/or error messages you see on your system could differ from what you see here. This is because my Linux distribution (and version), compiler/linker, library versions, and perhaps even CPU, may differ from yours. I built this on a x86_64 box running the Fedora 27/28 Linux distribution.

Similarly, we build the debug version of the `hello` program (again, ignoring the warning for now), and run it:

```
$ make hello_dbg
[...]
$ ./hello_dbg
Hello, Linux System Programming, World!
$
```

We use the powerful `objdump` utility to see the intermixed source-assembly-machine language of our program (`objdump's --source option switch -S, --source Intermix source code with disassembly`):

```
$ objdump --source ./hello_dbg
./hello_dbg:     file format elf64-x86-64

Disassembly of section .init:

0000000000400400 <_init>:
  400400:    48 83 ec 08              sub    $0x8,%rsp
```

```
[...]

int main(void)
{
  400527:    55                          push   %rbp
  400528:    48 89 e5                    mov    %rsp,%rbp
  40052b:    48 83 ec 10                 sub    $0x10,%rsp
    int a;

    printf("Hello, Linux System Programming, World!\n");
  40052f:    bf e0 05 40 00              mov    $0x4005e0,%edi
  400534:    e8 f7 fe ff ff              callq  400430 <puts@plt>
    a = 5;
  400539:    c7 45 fc 05 00 00 00        movl   $0x5,-0x4(%rbp)
    exit(0);
  400540:    bf 00 00 00 00              mov    $0x0,%edi
  400545:    e8 f6 fe ff ff              callq  400440 <exit@plt>
  40054a:    66 0f 1f 44 00 00           nopw   0x0(%rax,%rax,1)

[...]

$
```

The exact assembly and machine code you see on your system will, in all likelihood, differ from what you see here; this is because my Linux distribution (and version), compiler/linker, library versions, and perhaps even CPU, may differ from yours. I built this on a x86_64 box running Fedora Core 27.

Alright. Let's take the line of source code `a = 5;` where, `objdump` reveals the corresponding machine and assembly language:

```
    a = 5;
  400539:    c7 45 fc 05 00 00 00        movl   $0x5,-0x4(%rbp)
```

We can now clearly see the following:

C source	Assembly language	Machine instructions
`a = 5;`	`movl $0x5,-0x4(%rbp)`	`c7 45 fc 05 00 00 00`

So, when the process runs, at some point it will fetch and execute the machine instructions, producing the desired result. Indeed, that's exactly what a programmable computer is designed to do!

Though we have shown examples of displaying (and even writing a bit of) assembly and machine code for the Intel CPU, the concepts and principles behind this discussion hold up for other CPU architectures, such as ARM, PPC, and MIPS. Covering similar examples for all these CPUs goes beyond the scope of this book; however, we urge the interested reader to study the processor datasheet and ABI, and try it out.

Accessing a register's content via inline assembly

Now that we've written a simple C program and seen its assembly and machine code, let's move on to something a little more challenging: a C program with inline assembly to access the contents of a CPU register.

Details on assembly-language programming are outside the scope of this book; refer to the *Further reading* section on the GitHub repository.

x86_64 has several registers; let's just go with the ordinary RCX register for this example. We do make use of an interesting trick: the x86 ABI calling convention states that the return value of a function will be the value placed in the accumulator, that is, RAX for the x86_64. Using this knowledge, we write a function that uses inline assembly to place the content of the register we want into RAX. This ensures that this is what it will return to the caller!

Assembly micro-basics includes the following:

```
at&t syntax:
       movq <src_reg>, <dest_reg>
Register        : prefix name with %
Immediate value : prefix with $
```

For more, see the *Further reading* section on the GitHub repository.

Let's take a look at the following code:

```
$ cat getreg_rcx.c
/*
 * getreg_rcx.c
 *
 ************************************************************
 * This program is part of the source code released for the book
```

```
 *  "Linux System Programming"
 *  (c) Kaiwan N Billimoria
 *  Packt Publishers
 *
 * From:
 *  Ch 1 : Linux System Architecture
 ************************************************************
 * Inline assembly to access the contents of a CPU register.
 * NOTE: this program is written to work on x86_64 only.
 */
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

typedef unsigned long u64;

static u64 get_rcx(void)
{
    /* Pro Tip: x86 ABI: query a register's value by moving its value
into RAX.
     * [RAX] is returned by the function! */
    __asm__ __volatile__(
            "push %rcx\n\t"
            "movq $5, %rcx\n\t"
            "movq %rcx, %rax");
                   /* at&t syntax: movq <src_reg>, <dest_reg> */
    __asm__ __volatile__("pop %rcx");
}

int main(void)
{
    printf("Hello, inline assembly:\n [RCX] = 0x%lx\n",
                get_rcx());
    exit(0);
}
$ gcc -Wall -Wextra getreg_rcx.c -o getreg_rcx
getreg_rcx.c: In function ‘get_rcx':
getreg_rcx.c:32:1: warning: no return statement in function returning
non-void [-Wreturn-type]
 }
 ^
$ ./getreg_rcx
Hello, inline assembly:
 [RCX] = 0x5
$
```

There; it works as expected.