# 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

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.




# Accessing a control register's content via inline

## assembly

Among the many fascinating registers on the x86_64 processor, there happen to be six control registers, named CR0 through CR4, and CR8. There's really no need to delve into detail regarding them; suffice it to say that they are crucial to system control.

For the purpose of an illustrative example, let's consider the CR0 register for a moment. Intel's manual states: CR0—contains system control flags that control operating mode and states of the processor.

> Intel's manuals can be downloaded conveniently as PDF documents from here (includes the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B and 3C): System Programming Guide):
>
> https://software.intel.com/en-us/articles/intel-sdm

Clearly, CR0 is an important register!
We modify our previous program to access and display its content (instead of the ordinary RCX register). The only relevant code (which has changed from the previous program) is the function that queries the CR0 register value:

```
static u64 get_cr0(void)
{
/* Pro Tip: x86 ABI: query a register's value by moving it's value into RAX.
* [RAX] is returned by the function! */
__asm__ __volatile__("movq %cr0, %rax");
/* at&t syntax: movq <src_reg>, <dest_reg> */
}
```

Build and run it:

```
$ make getreg_cr0
[...]
$ ./getreg_cr0
Segmentation fault (core dumped)
$
```

It crashes!

Well, what happened here? Read on.





## CPU privilege levels

As mentioned earlier in this chapter, the essential job of the CPU is to read in machine instructions from memory, decipher, and execute them. In the early days of computing, this is pretty much all the processor did. But then, engineers, thinking deeper on it, realized that there is a critical issue with this: if a programmer can feed an arbitrary stream of machine instructions to the processor, which it, in turn, blindly and obediently executes, herein lies scope to do damage, to hack the machine!

How? Recall from the previous section the Intel processor's CR0 control register: Contains system control flags that control operating mode and states of the processor. If one has unlimited (read/write) access to the CR0 register, one could toggle bits that could do the following:

- Turn hardware paging on or off
- Disable the CPU cache
- Change caching and alignment attributes
- Disable WP (write protect) on memory (technically, pages) marked as read only by the OS

Wow, a hacker could indeed wreak havoc. At the very least, only the OS should be allowed this kind of access.

Precisely for reasons such as the security, robustness, and correctness of the OS and the hardware resources it controls, all modern CPUs include the notion of privilege levels.

The modern CPU will support at least two privilege levels, or modes, which are generically called the following:

- Supervisor
- User





You need to understand that code, that is, machine instructions, runs on the CPU at a given privilege level or mode. A person designing and implementing an OS is free to exploit the processor privilege levels. This is exactly how modern OSes are designed. Take a look at the following table Generic CPU Privilege Levels:

**Privilege level or mode name Privilege level Purpose Terminology** Supervisor High OS code runs here kernel-space User Low Application code runs here user-space (or userland)

Table 1: Generic CPU Privilege Levels

## Privilege levels or rings on the x86

To understand this important concept better, let's take the popular x86 architecture as a real example. Right from the i386 onward, the Intel processor supports four privilege levels or rings: **Ring 0**, **Ring 1**, **Ring 2**, and **Ring 3**. On the Intel CPU's, this is how the levels work:

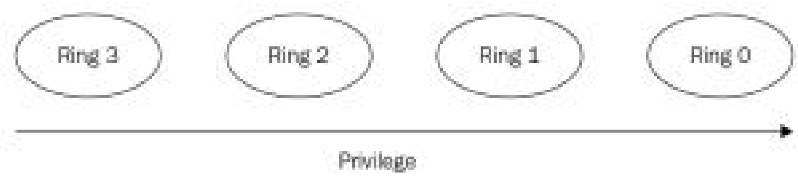


Figure 1: CPU ring levels and privilege

Let's visualize this *Figure 1* in the form of a *Table 2: x86 privilege or ring levels*:

| **Privilege or ring level** | **Privilege** | **Purpose** |
|---|---|---|
| Ring 0 | Highest | OS code runs here |
| Ring 1 | < ring 0 | <Unused> |
| Ring 2 | < ring 1 | <Unused> |
| Ring 3 | Lowest | Application code runs here (userland) |

Table 2: x86 privilege or ring levels





Originally, ring levels 1 and 2 were intended for device drivers, but modern OSes typically run driver code at ring 0 itself. Some hypervisors (VirtualBox being one) used to use Ring 1 to run the guest kernel code; this was the case earlier when no hardware virtualization support was available (Intel VT-x, AMD SV).

The ARM (32-bit) processor has seven modes of execution; of these, six are privileged, and only one is the non-privileged mode. On ARM, generically, the equivalent to Intel's Ring 0 is Supervisor (SVC) mode, and the equivalent to Intel's Ring 3 is User mode.

For interested readers, there are more links in the *Further* *reading* section on the GitHub repository.

The following diagram clearly shows of all modern OSes (Linux, Unix, Windows, and macOS) running on an x86 processor exploit processor-privilege levels:

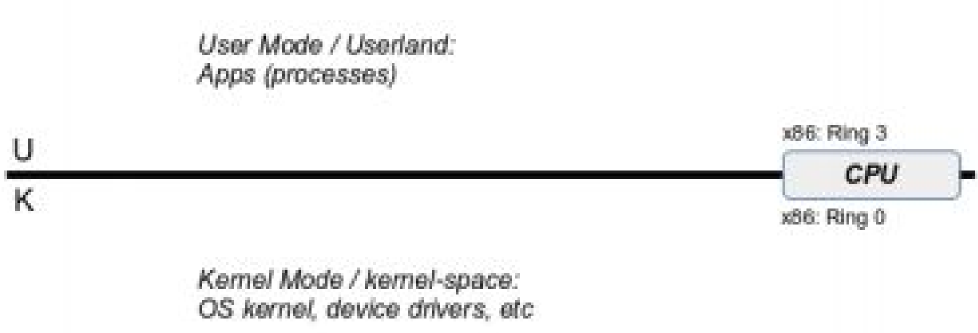


Figure 2: User-Kernel separation

Importantly, the processor ISA assigns every machine instruction with a privilege level or levels at which they are allowed to be executed. A machine instruction that is allowed to execute at the user privilege level automatically implies it can also be executed at the Supervisor privilege level. This distinguishing between what can and cannot be done at what mode also applies to register access.

To use the Intel terminology, the **Current Privilege Level** (**CPL**) is the privilege level at which the processor is currently executing code.





For example, that on a given processor shown as follows:

The foo1 machine instruction has an allowed privilege level of Supervisor (or Ring 0 for x86)

The foo2 machine instruction has an allowed privilege level of User (or Ring 3 for x86)

So, for a running application that executes these machine instructions, the following table emerges:

| **Machine instruction** | **Allowed-at mode** | **CPL (current privilege level)** | **Works?** |
|---|---|---|---|
| foo1 | Supervisor (0) | 0 | Yes |
| | | 3 | No |
| foo2 | User (3) | 0 | Yes |
| | | 3 | Yes |

Table 3: Privilege levels – an example

> So, thinking about it, foo2 being allowed at User mode would also be allowed to execute with any CPL. In other words, if the CPL <= allowed privilege level, it works, otherwise it does not.

When one runs an application on, say, Linux, the application runs as a process (more on this later). But what privilege (or mode or ring) level does the application code run at? Refer to the preceding table: User Mode (Ring 3 on x86).

Aha! So now we see. The preceding code example, `getreg_rcx.c`, worked because it attempted to access the content of the general-purpose `RCX` register, which is allowed in User Mode (Ring 3, as well as at the other levels, of course)!

But the code of `getreg_cr0.c` failed; it crashed, because it attempted to access the content of the `CR0` control register, which is disallowed in User Mode (Ring 3), and allowed only at the Ring 0 privilege! Only OS or kernel code can access the control
registers. This holds true for several other sensitive assembly-language instructions as well. This approach makes a lot of sense.

> Technically, it crashed because the processor raised a **General Protection Fault** (**GPF**).





# Linux architecture

The Linux system architecture is a layered one. In a very simplistic way, but ideal to start on our path to understanding these details, the following diagram illustrates the Linux system architecture:

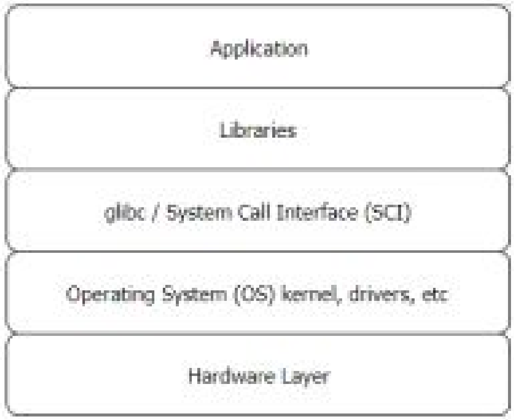

Figure 3: Linux – Simplified layered architecture

Layers help, because each layer need only be concerned with the layer directly above and below it. This leads to many advantages:

Clean design, reduces complexity
Standardization, interoperability
Ability to swap layers in and out of the stack
Ability to easily introduce new layers as required

On the last point, there exists the FTSE. To quote directly from Wikipedia:

The "**fundamental theorem of software engineering** (**FTSE**)" is a term originated by Andrew Koenig to describe a remark by Butler Lampson attributed to the late David J. Wheeler

We can solve any problem by introducing an extra level of indirection.




Now that we understand the concept of CPU modes or privilege levels, and how modern OSes exploit them, a better diagram (expanding on the previous one) of the Linux system architecture would be as follows:

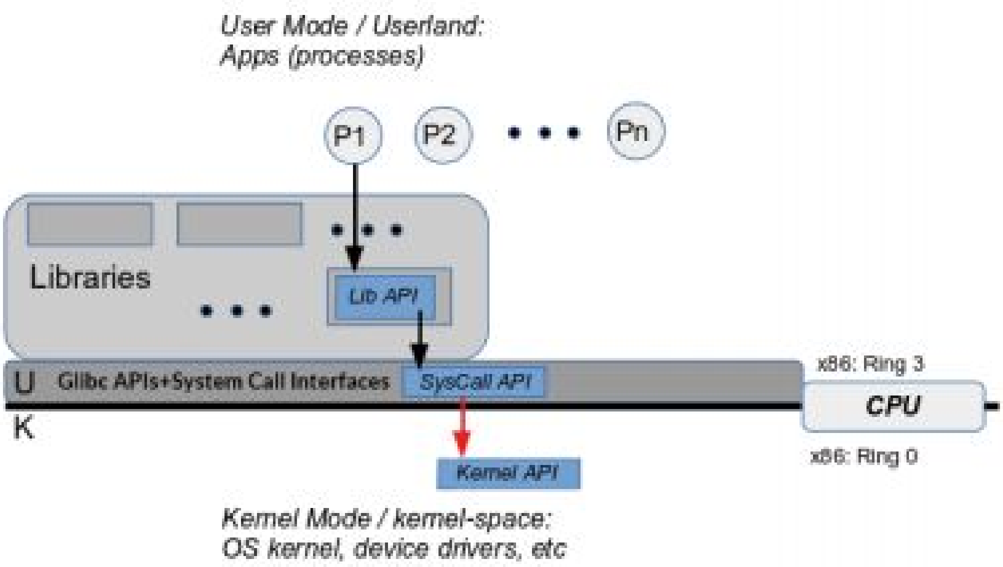


Figure 4: Linux system architecture

In the preceding diagram, **P1**, **P2**, …, **Pn** are nothing but userland processes (Process 1, Process 2) or in other words, running applications. For example, on a Linux laptop, we might have the vim editor, a web browser, and terminal windows (gnome terminal) running.

## Libraries

Libraries, of course, are archives (collections) of code; as we well know, using libraries helps tremendously with code modularity, standardization, preventing the reinvent the-wheel syndrome, and so on. A Linux desktop system might have libraries numbering in the hundreds, and possibly even a few thousand!

The classic K&R hello, world C program uses the printf API to write the string to the display:

```
printf(“hello, world\n”);
```





Obviously, the code of printf is not part of the hello, world source. So where does it come from? It's part of the standard C library; on Linux, due to its GNU origins, this library is commonly called **GNU libc** (**glibc**).

Glibc is a critical and required component on a Linux box. It not only contains the usual standard C library routines (APIs), it is, in fact, the programming interface to the operating system! How? Via its lower layer, the system calls.

## System calls

System calls are actually kernel functionality that can be invoked from userspace via glibc stub routines. They serve a critical function; they connect userspace to kernel space. If a user program wants to request something of the kernel (read from a file, write to the network, change a file's permissions), it does so by issuing a system call. Therefore, system calls are the only legal entry point to the kernel. There is no other way for a user-space process to invoke the kernel.

> For a list of all the available Linux system calls, see section 2 of the man pages (https://linux.die.net/man/2/). One can also do: man 2 syscalls to see the man page on all supported system calls

Another way to think of this: the Linux kernel internally has literally thousands of APIs (or functions). Of these, only a small fraction are made visible or available, that is, exposed, to userspace; these exposed kernel APIs are system calls! Again, as an approximation, modern Linux glibc has around 300 system calls.

> On an x86_64 Fedora 27 box running the 4.13.16-302.fc27.x86_64 kernel, there are close to 53,000 kernel APIs!

Here is the key thing to understand: system calls are very different from all other (typically library) APIs. As they ultimately invoke kernel (OS) code, they have the ability to cross the user-kernel boundary; in effect, they have the ability to switch from normal unprivileged User mode to completely privileged Supervisor or kernel mode!





How? Without delving into the gory details, system calls essentially work by invoking special machine instructions that have the built-in ability to switch the processor mode from User to Supervisor. All modern CPU ABIs will provide at least one such machine instruction; on the x86 processor, the traditional way to implement system calls is to use the special int 0x80 machine instruction. Yes, it is indeed a software interrupt (or trap). From Pentium Pro and Linux 2.6 onward, the sysenter/syscall machine instructions are used. See the *Further reading* section on the GitHub repository.

From the viewpoint of the application developer, a key point regarding system calls is that system calls appear to be regular functions (APIs) that can be invoked by the developer; this design is deliberate. The reality: the system call APIs that one invokes – such as open(), read(), chmod(), dup(), and write() – are merely stubs. They are a neat mechanism to get at the actual code that is in the kernel (getting there involves populating a register the accumulator on x86 – with the system call number, and passing parameters via other general-purpose registers) to execute that kernel code path, and return back to user mode when done. Refer to the following table:

| CPU | Machine instruction(s) used to trap to Supervisor (kernel) Mode from User Mode | Allocated Register for system call number |
|---|---|---|
| x86[_64] | int 0x80 or syscall | EAX / RAX |
| ARM | swi / svc | R0 to R7 |
| Aarch64 | svc | X8 |
| MIPS | syscall | $v0 |

Table 4: System calls on various CPU Architectures for better understanding

# Linux – a monolithic OS

Operating systems are generally considered to adhere to one of two major architectural styles: monolithic or microkernel.

Linux is decidedly a monolithic OS.





## What does that mean?

The English word monolith literally means a large single upright block of stone:  Figure 5: Corinthian columns – they're monolithic!





On the Linux OS, applications run as independent entities called **processes**. A process may be single-threaded (original Unix) or multithreaded. Regardless, for now, we will consider the process as the unit of execution on Linux; a process is defined as an instance of a program in execution.

When a user-space process issues a library call, the library API, in turn, may or may not issue a system call. For example, issuing the atoi(3) API does not cause glibc to issue a system call as it does not require kernel support to implement the conversion of a string into an integer. <api-name>(n) ; n is the man page section.

To help clarify these important concepts, let's check out the famous and classic K&R Hello, World C program again:

```
#include <stdio.h>
main()
{
 printf("hello, world\n");
 }
```

Okay, that should work. Indeed it does.
But, the question is, how exactly does the printf(3) API write to the monitor device?

The short answer: it does not.
The reality is that printf(3) only has the intelligence to format a string as specified; that's it. Once done, printf actually invokes the write(2) API – a system call. The write system call does have the ability to write the buffer content to a special device file – the monitor device, seen by write as stdout. Go back to our discussion regarding *The Unix philosophy in a nutshell* : if it's not a process, it's a file! Of course, it gets really complex under the hood in the kernel; to cut a long story short, the kernel code of write ultimately switches to the correct driver code; the device driver is the only component that can directly work with peripheral hardware. It performs the actual write to the monitor, and return values propagate all the way back to the application.





In the following diagram, **P** is the hello, world process at runtime:

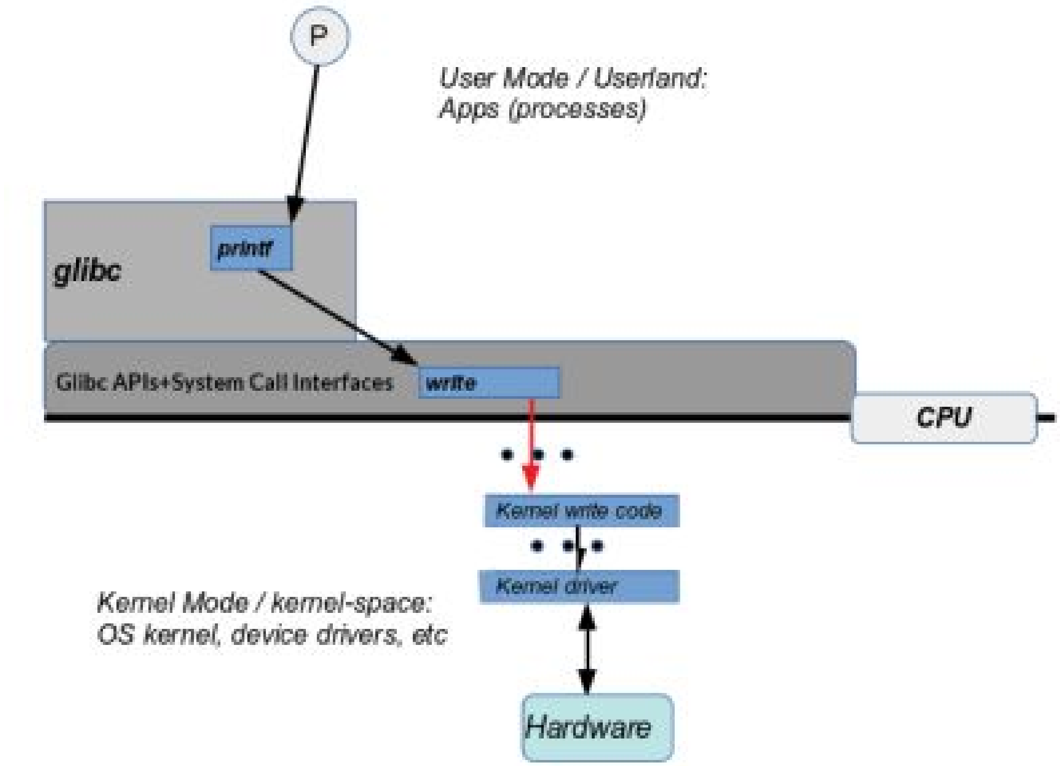


Fig 6: Code flow: printf-to-kernel

Also, from the diagram, we can see that **glibc** is considered to consist of two parts:

> **Arch-independent glibc**: The regular libc APIs (such as [s|sn|v]printf, memcpy, memcmp, atoi)
>
> **Arch-dependent glibc**: The **system call** stubs

> Here, by arch, we mean CPU.
> Also the ellipses (**...**) represent additional logic and processing within kernel-space that we do not show or delve into here.

Now that the code flow path of hello, world is clearer, let's get back to the monolithic stuff!





It's easy to assume that it works this way:

1. The hello, world app (process) issues the printf(3) library call. 2. printf issues the write(2) system call.
3. We switch from User to Supervisor (kernel) Mode.
4. The kernel takes over – it writes hello, world onto the monitor. 5.

Switch back to non-privileged User Mode.

Actually, that's NOT the case.

The reality is, in the monolithic design, there is no kernel; to word it another way, the kernel is actually part of the process itself. It works as follows:

1. The hello, world app (process) issues the printf(3) library call. 2. printf issues the write(2) system call.
3. The process invoking the system call now switches from User to Supervisor (kernel) Mode.
4. The process runs the underlying kernel code, the underlying device driver code, and thus, writes hello, world onto the monitor!
5. The process is then switched back to non-privileged User Mode.

To summarize, in a monolithic kernel, when a process (or thread) issues a system call, it switches to privileged Supervisor or kernel mode and runs the kernel code of the system call (working on kernel data). When done, it switches back to unprivileged User mode and continues executing userspace code (working on user data).

This is very important to understand:

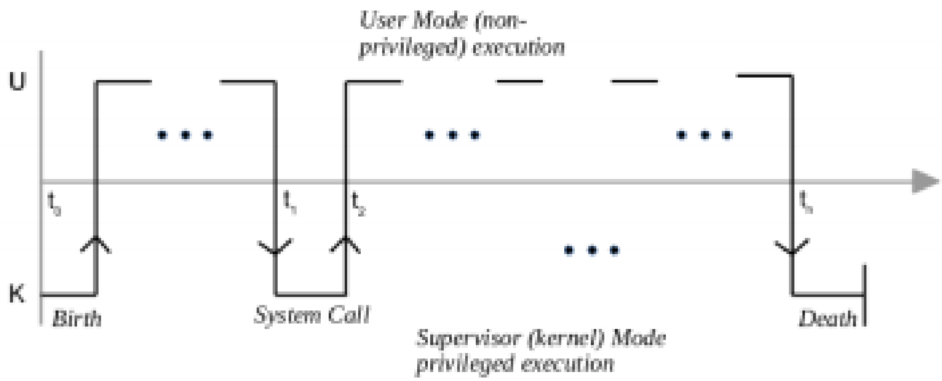


Fig 7: Life of a process in terms of privilege modes





The preceding diagram attempts to illustrate that the **X axis** is the **timeline**, and the **Y axis** represents **User Mode** (at the top) and **Supervisor (kernel) Mode** (at the bottom):

**time $t_0$**: A process is born in kernel mode (the code to create a process is within the kernel of course). Once fully born, it is switched to User (non privileged) Mode and it runs its userspace code (working on its userspace data items as well).

**time $t_1$**: The process, directly or indirectly (perhaps via a library API), invokes a system call. It now traps into kernel mode (refer the table *System*

*Calls on CPU Architectures* shows the machine instructions depending on the CPU to do so) and executes kernel code in privileged Supervisor Mode (working on kernel data items as well).

**time $t_2$**: The system call is done; the process switches back to non-privileged User Mode and continues to execute its userspace code. This process continues, until some point in the future.

**time $t_n$**: The process dies, either deliberately by invoking the exit API, or it is killed by a signal. It now switches back to Supervisor Mode (as the exit(3) library API invokes the _exit(2) system call), executes the kernel code of _exit(), and terminates.

In fact, most modern operating systems are monolithic (especially the Unix-like ones).

> Technically, Linux is not considered 100 percent monolithic. It's considered to be mostly monolithic, but also modular, due to the fact that the Linux kernel supports modularization (the plugging in and out of kernel code and data, via a technology called **Loadable Kernel Modules** (**LKMs**)).
> Interestingly, MS Windows (specifically, from the NT kernel onward) follows a hybrid architecture that is both monolithic and microkernel.

# Execution contexts within the kernel

Kernel code always executes in one of two contexts:

- Process
- Interrupt





> It's easy to get confused here. Remember, this discussion applies to the context in which kernel code executes, not userspace code.

## Process context

Now we understand that one can invoke kernel services by issuing a system call. When this occurs, the calling process runs the kernel code of the system call in kernel mode. This is termed **process context** – kernel code is now running in the context of the process that invoked the system call.

Process context code has the following attributes:

- Always triggered by a process (or thread) issuing a system call
- Top-down approach
- Synchronous execution of kernel code by a process

## Interrupt context

At first glance, there appears to be no other way that kernel code executes. Well, think about this scenario: the network receive path. A network packet destined for your Ethernet MAC address arrives at the hardware adapter, the hardware detects that it's meant for it, collects it, and buffers it. It now must let the OS know; more technically, it must let the **Network Interface Card** (**NIC**) device driver know, so that it can fetch and process packets as they arrive. It kicks the NIC driver into action by asserting a hardware interrupt.

Recall that device drivers reside in kernel-space, and therefore their code runs in Supervisor or kernel Mode. The (kernel privilege) driver code **Interrupt service routine** (**ISR**) now executes, fetches the packet, and sends it up the OS network protocol stack for processing.

The NIC driver's ISR code is kernel code, and it is has run but in what context? It's obviously not in the context of any particular process. In fact, the hardware interrupt probably interrupted some process. Thus, we just call this *interrupt context*.





The interrupt context code has the following attributes:

- Always triggered by a hardware interrupt (not a software interrupt, fault or exception; that's still process context)
- Bottom-up approach
- Asynchronous execution of kernel code by an interrupt

> If, at some point, you do report a kernel bug, it helps if you point out the execution context.

Technically, within interrupt context, we have further distinctions, such as hard-IRQs and softirqs, bottom halves, and tasklets. However, this discussion goes beyond the scope of this book.

# Summary

This chapter started by explaining the Unix design philosophy, including the central principles or pillars of the Unix philosophy, design, and architecture. We then described the Linux system architecture, where we covered the meaning of CPU-ABI (Application Binary Interface), ISA, and toolchain (using `objdump` to disassemble a simple program, and accessing CPU registers with inline assembly). CPU privilege levels and their importance in the modern OS were discussed, leading in to the Linux system architecture layers – application, libraries, system calls, and the kernel. The chapter finished with a discussion on how Linux is a monolithic OS and then explored kernel execution contexts.

In the next chapter, the reader will delve into the mysteries of, and get a solid grasp of, virtual memory – what exactly it means, why it's in all modern OSes, and the key benefits it provides. We will discuss relevant details of the making of process virtual address space.



# 2

# Virtual Memory

Coming back to this chapter, we will look at the meaning and purpose of **virtual memory** (**VM**) and, importantly, why it is a key concept and required one. We will cover the meaning and importance of VM, paging and address-translation, the benefits of using VM, the memory layout of a process in execution, and the internal layout of a process as seen by the kernel. We shall also delve into what segments make up the process virtual address space. This knowledge is indispensable in difficult-to-debug situations.

In this chapter, we will cover the following topics:

- Virtual memory
- Process virtual address space

# 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.

# Virtual memory

Modern operating systems are based on a memory model called VM**.** This includes Linux, Unixes, MS Windows, and macOS. Truly understanding how a modern OS works under the hood requires a deep understanding of VM and memory management – not topics we delve into in intricate detail in this book; nevertheless, a solid grasp of VM concepts is critical for Linux system developers.





# No VM – the problem

Let's imagine for a moment that VM, and all the complex baggage it lugs around, does not exist. So, we're working on a (fictional) pure flat physical memory platform with, say, 64 MB RAM. This is actually not that unusual – most old OSes (think DOS) and even modern **Real-Time Operating Systems** (RTOSes) operate this way:

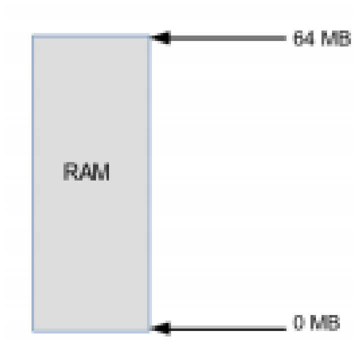


Figure 1: Flat physical address space of 64 MB

Obviously, everything that runs on this machine must share this physical memory space: the OS, device drivers, libraries, and applications. We might visualize it this way (of course, this is not intended to reflect an actual system – it's just a highly simplified example to help you understand things): one OS, several device drivers (to drive the hardware peripherals), a set of libraries, and two applications. The physical memory map (not drawn to scale) of this fictional (64 MB system) platform might look like this:

**Object Space taken Address range**

Operating system (OS) 3 MB 0x03d0 0000 - 0x0400 0000 Device Drivers 5 MB 0x02d0 0000 – 0x0320 0000 Libraries 10 MB 0x00a0 0000 – 0x0140 0000 Application 2 1 MB 0x0010 0000 – 0x0020 0000 Application 1 0.5 MB 0x0000 0000

– 0x0008 0000 Overall Free Memory 44.5 MB <various>

Table 1: The physical memory map





The same fictional system is represented in the following

diagram:

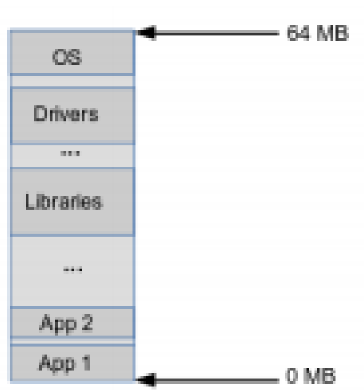


Fig 2: The physical memory map of our fictional 64 MB system

Normally, of course, the system will undergo rigorous testing before release and will perform as expected; except, there's this thing you might have heard of in our industry called bugs. Yes, indeed.

But let's imagine a dangerous bug creeps into Application 1, say, within the use of the ubiquitous memcpy(3) glibc API, due to either of the following:

- Inadvertent programming errors
- Deliberate malicious intent

As a quick reminder, the usage of the memcpy library API is shown as

follows: `void *memcpy(void *dest, const void *src, size_t n)`.

## Objective

This C program snippet as follows intends to copy some memory, say 1,024 bytes, using the usual memcpy(3) glibc API, from a source location 300 KB into the program to a destination location 400 KB into the program. As Application 1 is the program at the low end of physical memory (see the preceding memory map), it starts at the 0x0 physical offset.





We understand that on a modern OS nothing will start at address 0x0; that's the canonical NULL memory location! Keep in mind that this is just a fictional example for learning purposes

First, let's see the correct usage case.

Refer to the following pseudocode:

```
phy_offset = 0x0;
src = phy_offset + (300*1024); /* = 0x0004 b000 */ dest = phy_offset +
(400*1024); /* = 0x0006 4000 */ n = 1024;
memcpy(dest, src, n);
```

The effect of the preceding code is shown in the following diagram:

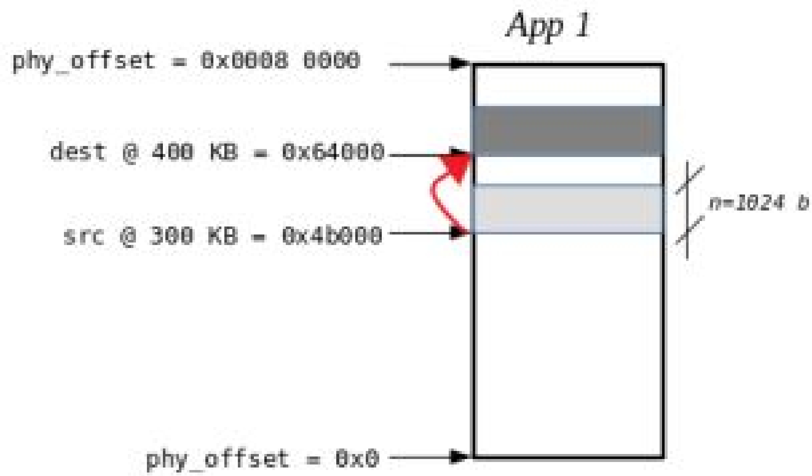


Fig 3: Zoomed into App 1: the correct memcpy()

As can be seen in the preceding diagram, this works! The (big) arrow shows the copy path from source to destination, for 1,024 bytes. Great.

Now for the buggy case.

All remains the same, except that this time, due to a bug (or malicious intent), the dest pointer is modified as follows:

```
phy_offset = 0x0;
src = phy_offset + (300*1024); /* = 0x0004 b000 */ dest = phy_offset +
(400*1024*156); /* = 0x03cf 0000 !BUG! */ n = 1024;
memcpy(dest, src, n);
```

The destination location is now around 64 KB (0x03cf0000 – 0x03d00000) into the operating system! The best part: the code itself does not fail. memcpy() does its job. Of course, now the OS is probably corrupted and the entire system will (eventually) crash.

Note that the intent here is not to debug the cause (we know); the intent here is to clearly realize that, in spite of this bug, memcpy succeeds.
How come? This is because we are programming in C – we are free to read and write physical memory as we wish; inadvertent bugs are our problem, not the language's!

So what now? Ah, this is one of the key reasons why VM systems came into existence.

# Virtual memory

Unfortunately, the term **virtual memory** (**VM**) is often misunderstood or hazily understood, at best, by a large proportion of engineers. In this section, we attempt to clarify what this term and its associated terminologies (such as memory pyramid, addressing, and paging) really mean; it's important for developers to clearly understand this key area.

First, what is a process?

A process is an instance of a program in execution.
A program is a binary executable file: a dead, disk object. For example, take the cat program:

```
$ ls -l /bin/cat
-rwxr-xr-x 1 root root 36784 Nov 10 23:26 /bin/cat $
```

When we run cat it becomes a live runtime schedulable entity, which, in the Unix universe, we call a process.

In order to understand deeper concepts clearly, we start with a small, simple, and fictional machine. Imagine it has a microprocessor with 16 address lines. Thus, it's easy to see, it will have access to a total potential memory space (or address space) of $2^{16} = 65,536$ bytes = 64 KB:

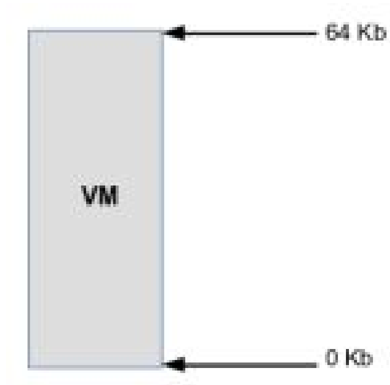


Fig 4: Virtual memory of 64 KB

But what if the physical memory (RAM) on the machine is a lot less, say, 32 KB? Clearly, the preceding diagram depicts virtual memory, not physical. Meanwhile, physical memory (RAM) looks as follows:

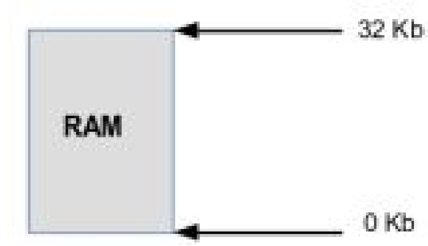


Fig 5: Physical memory of 32 KB

Still, the promise made by the system to every process alive: every single process will have available to it the entire virtual address space, that is, 64 KB. Sounds absurd, right? Yes, until one realizes that memory is more than just RAM; in fact, memory is viewed as a hierarchy – what's commonly referred to as the memory pyramid:

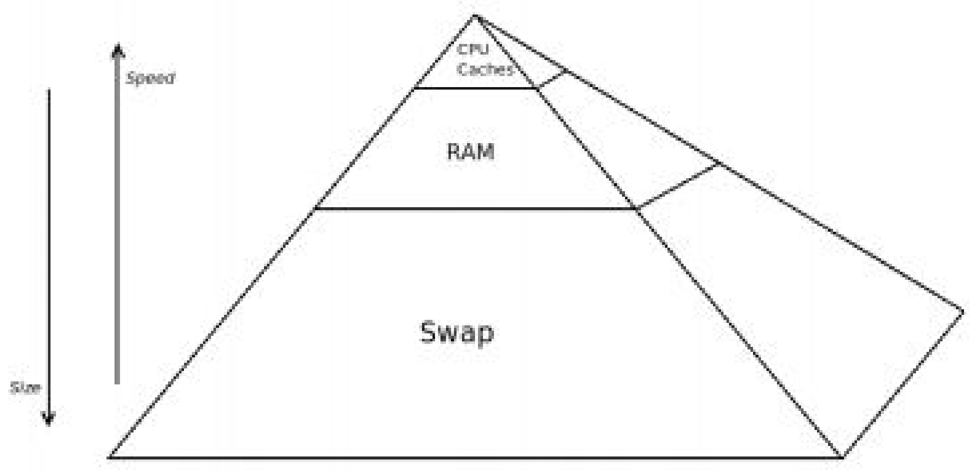


Fig 6: The Memory pyramid

As with life, everything's a trade-off. Toward the apex of the pyramid, we gain in **Speed** at the cost of size; toward the bottom of the pyramid, it's inverted: **Size** at the cost of speed. One could also consider CPU registers to be at the very apex of the pyramid; as its size is almost insignificant, it has not been shown.

> *Swap* is a filesystem type – a raw disk partition is formatted as swap upon system installation. It's treated as second-level RAM by the OS. When the OS runs out of RAM, it uses swap. As a rough heuristic, system administrators sometimes configure the size of the swap partition to be twice that of available RAM.





To help quantify this, according to *Computer Architecture*, *A Quantitative Approach*, *5th Ed*, by Hennessy & Patterson, fairly typical numbers follow:

| **Type** | **CPU registers** | **CPU caches** | | | **RAM** | **Swap/storage** |
|---|---|---|---|---|---|---|
| | | L1 | L2 | L3 | | |
| Server | 1000 bytes | 64 KB | 256 KB | 2 - 4 MB | 4 - 16 GB | 4 - 16 TB |
| | 300 ps | 1 ns | 3 - 10 ns | 10 - 20 ns | 50 - 100 ns | 5 - 10 ms |
| Embedded | 500 bytes | 64 KB | 256 KB | - | 256 - 512 MB | 4 - 8 GB Flash |
| | 500 ps | 2 ns | 10 - 20 ns | - | 50 - 100 ns | 25 - 50 us |

Table 2: Memory hierarchy numbers

Many (if not most) embedded Linux systems do not support a swap partition; the reason is straightforward: embedded systems mostly use flash memory as the secondary storage medium (not a traditional SCSI disk as do laptops, desktops, and servers). Writing to a flash chip wears it out (it has limited erase-write cycles); hence, embedded-system designers would rather sacrifice swap and just use RAM. (Please note that the embedded system can still be VM based, which is the usual case with Linux and Win-CE, for example).

The OS will do its best to keep the working set of pages as high up the pyramid as is possible, optimizing performance.

It's important for the reader to note that, in the sections that follow, while this book attempts to explain some of the inner workings of advanced topics such as VM and addressing (paging), we quite deliberately do not paint a complete, realistic, real-world view.

The reason is straightforward: the deep and gory technical details are well beyond the scope of this book. So, the reader should keep in mind that several of the following areas are explained in concept and not in actuality. The *Further reading* section provides references for readers who are interested in going deeper into these matters. Refer it on the GitHub repository.





# Addressing 1 – the simplistic flawed approach

Okay, now to the memory pyramid; even if we agree that virtual memory is now a possibility, a key and difficult hurdle to overcome remains. To explain this, note that every single process that is alive will occupy the entire available **virtual address space** (**VAS**). Thus, each process overlaps with every other process in terms of VAS. But how would this work? It wouldn't, by itself. In order for this elaborate scheme to work, the system has to somehow map every virtual address in every process to a physical address! Refer to the following mapping of virtual address to physical address:

**Process P:virtual address (va) → RAM:physical address (pa)**

So, the situation is something like this now:

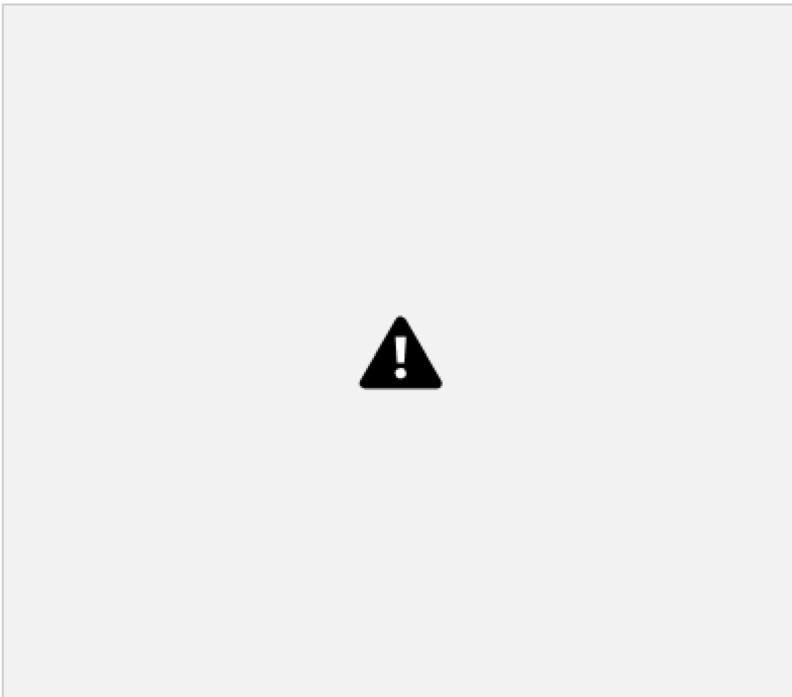

Fig 7: Processes containing virtual addresses





Processes **P1**, **P2**, and **Pn**, are alive and well in VM. Their virtual address spaces cover 0 to 64 KB and overlap each other. Physical memory, **RAM**, of 32 KB is present on this (fictional) system.

As an example, two virtual addresses for each process are shown in the following format:

P'r':va'n'; where r is the process number and n is 1 and 2.

As mentioned earlier, the key now is to map each process's virtual addresses to physical addresses. So, we need to map the following:

**P1:va1 → P1:pa1**
**P1:va2 → P1:pa2**
**...**

**P2:va1 → P2:pa1**
**P2:va2 → P2:pa2**
**...**

**[...]**

**Pn:va1 → Pn:pa1**
**Pn:va2 → Pn:pa2**
**...**





We could have the OS perform this mapping; the OS would then maintain a mapping table per process to do so. Diagrammatically and conceptually it looks as follows:

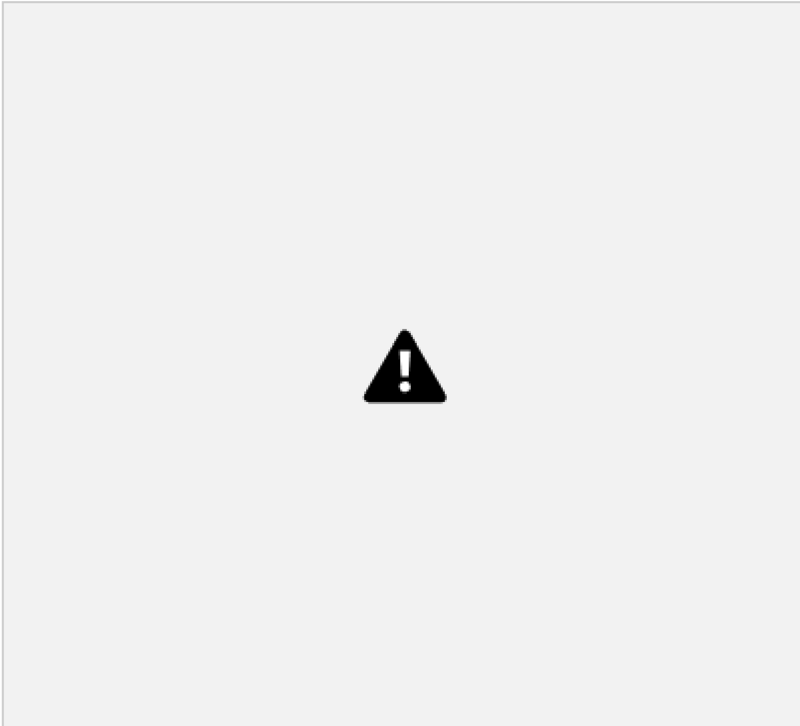

Fig 8: Direct mapping virtual addresses to physical RAM addresses

So that's it, then? Seems quite simple, actually. Well, no, it won't work in reality: to map all the possible virtual addresses per process to physical addresses in RAM, the OS would need to maintain a **va**-to-**pa** translation entry per address per process! That's too expensive, as each table would possibly exceed the size of physical memory, rendering the scheme useless.

A quick calculation reveals that we have 64KB virtual memory, that is, 65,536 bytes or addresses. Each of these virtual addresses need to be mapped to a physical address. So each process would require:

65536 * 2 = 131072 = 128 KB, for a mapping table. per process.





It gets worse in reality; the OS would need to store some metadata along with each address-translation entry; let's say 8 bytes of metadata. So now, each process would require:

65536 * 2 * 8 = 1048576 = 1 MB, for a mapping table. per process.

Wow, 1 megabyte of RAM per process! That's far too much (think of an embedded system); also, on our fictional system, there's a total of 32 KB of RAM. Whoops.

Okay, we can reduce this overhead by not mapping each byte but mapping each word; say, 4 bytes to a word. So now, each process would require:

(65536 * 2 * 8) / 4 = 262144 = 256 KB, for a mapping table. per process.

Better, but not good enough. If there are just 20 processes alive, we'd require 5 MB of physical memory to store just the mapping metadata. With 32 KB of RAM, we can't do that.

## Addressing 2 – paging in brief

To address (pun intended) this tricky issue, computer scientists came up with a solution: do not attempt to map individual virtual bytes (or even words) to their physical counterpart; it's far too expensive. Instead, carve up both physical and virtual memory space into blocks and map them.

A bit simplistically, there are broadly two ways to do this:

Hardware-segmentation
Hardware-paging

**Hardware-segmentation:** Carves up the virtual and physical address space into arbitrary-sized chunks called **segments**. The best example is Intel 32-bit processors.

**Hardware-paging:** Carves up the virtual and physical address space into equal-sized chunks called **pages**. Most real-world processors support hardware-paging, including Intel, ARM, PPC, and MIPS.

Actually it's not even up to the OS developer to select which scheme to use: the choice is dictated by the hardware MMU.





Again, we remind the reader: the intricate details are beyond the scope of this book. See the *Further reading* section on the GitHub repository.

Let's assume we go with the paging technique. The key takeaway is that we stop attempting to map all possible virtual addresses per process to physical addresses in

RAM, instead, we map virtual pages (just called pages) to physical pages (called page frames).

**Common Terminology**

**virtual address space** : **VAS**
Virtual page within the process VAS : page
Physical page in RAM : **page frame** (**pf**)

Does NOT work: **virtual address** (**va**) → **physical address** (**pa**)
Does work: (virtual) page → page frame

The left-to-right arrow represents the mapping.

As a rule of thumb (and the generally accepted norm), the size of a page is 4 kilobytes (4,096 bytes). Again, it's the processor **Memory Management Unit** (**MMU**) that dictates the page size.

So how and why does this scheme help?

Think about it for a moment; in our fictional machine, we've got: 64 KB of VM, that is, 64K/4K = 16 pages, and 32 KB of RAM, that is, 32K/4K = 8 page frames.

Mapping 16 pages to corresponding page frames requires a table of only 16 entries per process; this is viable!

As in our earlier calculations:
16 * 2 * 8 = 256 bytes, for a mapping table per process.

The very important thing, it bears repeating: we map (virtual) pages to (physical) page frames!