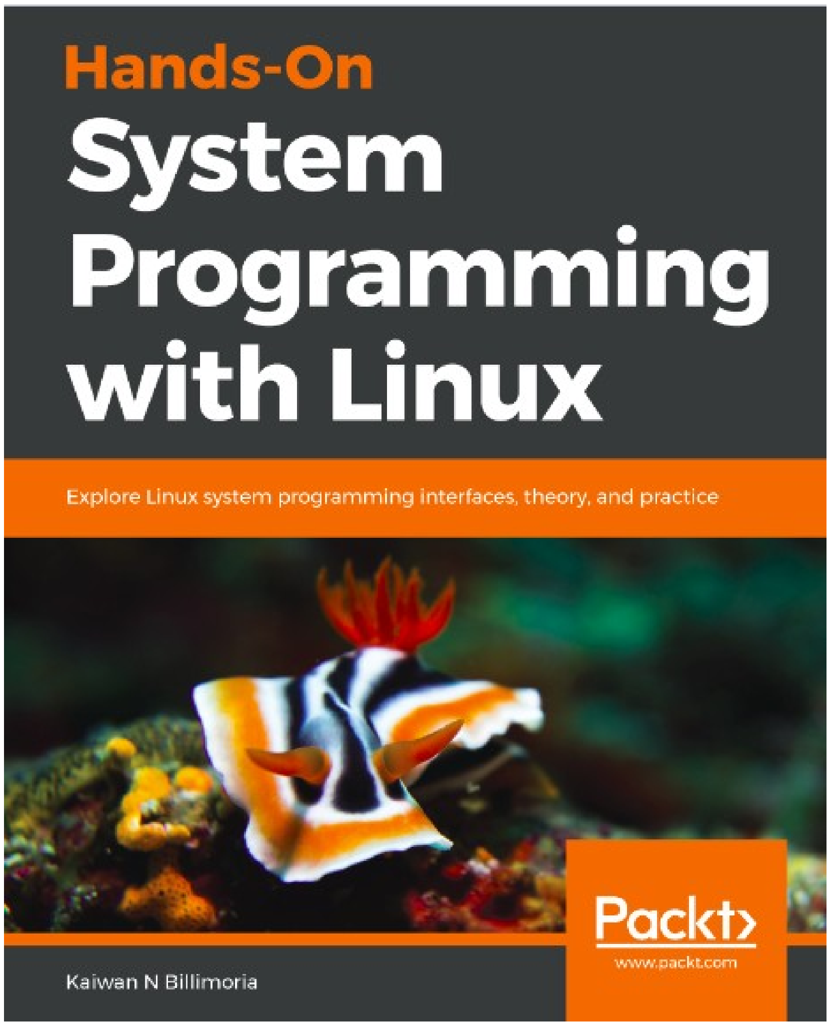
Hands-On System Programming with Linux, Kaiwan N Billimoria
Publisher: Packt

Look it up, order online: **On Amazon (Kindle / Paperback)** **On Packt**


Hands-On
System Programming with Linux
Explore Linux system programming interfaces, theory, and practice
Packt>
www.packt.com
Kaiwan N Billimoria

Book Description

The Linux OS and it's embedded and server applications are a critical component of today's key software infrastructure in a decentralized, networked universe. The industry demand for proficient Linux developers is only rising. This book aims to give the reader two things: a solid theoretical base and practical industry-relevant descriptions and code covering the Linux system programming domain. It delves into the art and science of Linux application programming - system architecture, process memory and management, signalling, timers, pthreads, file IO, shared libraries, and more.

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 tradeoffs made by experienced developers when using them and the rationale behind them. Troubleshooting tips and techniques round out the book's coverage.

By the end of this book, you will have both the essential conceptual design knowledge and hands-on experience working with Linux system programming interfaces.

What You Will Learn

- A solid grasp on the key theoretical underpinnings of the UNIX and Linux system architecture
- The what and why of modern OS's using Virtual Memory, dynamic memory APIs
- Process Management: concepts and APIs on process credentials, execution, creation, signalling
- How to effectively perform file IO, and use timers
- Multithreading: concepts and APIs on pthreads creation, management, cancellation, synchronization in detail

- Linux's CPU scheduling policies and how to exploit them via their API set
- Build and work with daemon processes
- How to effectively troubleshoot.

Amazon Reviews

- 5.0 out of 5 stars **approach to understand various basic programming concepts**

13 Dec

Format: Kindle Edition

This book is first of its kind to understand various programming techniques through best suitable coding practices. More importantly it reveals various OS concepts and underlying techniques of handling user space applications. It includes debugging techniques, trouble shooting and tracing tools etc which helps to understand and avoid common programming mistakes as well as helps to write well performing codes. This book is a best option to understand many important OS concepts like process, thread, CPU scheduling and signalling etc. Personally it helped me to get clarity in various operating system concepts and understand best programming practices.

- 5.0 out of 5 stars **ls on book for a linux geek**

10 Dec

Format: Kindle Edition

Covers fundamentals & key aspects – systems programming at the library and system call layers. Most important is that it covers the "why" and with a hands-on approach of the "how". Throughout the book there are useful real work examples. Deep dive into various key topics like Virtual Memory, memory debug tools like (valgrind, sanitizers), rules of fork(), cpu scheduling and much more. I highly recommend this book for linux programmers who want take a deep dive into linux system programming.

- 5.0 out of 5 stars **ands on.**

16 Janu

Format: Kindle Edition

A well organized book in system programming with Linux at library & system calls.
Talks about architecture, memory, signal, process, multi-threads, timers, file i/o.
Crisp and clear explanation with test codes, expected output, and explanation related the system behavior.
The example codes used are very well explained and in sync to understand the concepts easily.
This book has become a quick reference guide for me now.

Table of Contents

Preface

1: Linux System Architecture

Technical requirements

Linux and the Unix operating system

The Unix philosophy in a nutshell

        Everything is a process – if it's not a process, it's a file
        One tool to do one task
        Three standard I/O channels
            Word count
            cat
    Combine tools seamlessly
    Plain text preferred
    CLI, not GUI
    Modular, designed to be repurposed by others
    Provide mechanisms, not policies
        Pseudocode
    Linux system architecture
    Preliminaries
        The ABI
        Accessing a register's content via inline assembly
        Accessing a control register's content via inline assembly
        CPU privilege levels
            Privilege levels or rings on the x86
    Linux architecture
        Libraries
        System calls
        Linux – a monolithic OS
            What does that mean?
    Execution contexts within the kernel
        Process context
        Interrupt context
    Summary

2: Virtual memory
    Virtual memory
        No VM – the problem
            Objective
        Virtual memory
            Addressing 1 – the simplistic flawed approach
            Addressing 2 – paging in brief
                Paging tables – simplified
                Indirection
                Address-translation
        Benefits of using VM
        Process-isolation
        The programmer need not worry about physical memory
        Memory-region protection
        SIDEBAR :: Testing the memcpy() C program
    Process memory layout
        Segments or mappings
            Text segment
            Data segments
            Library segments
            Stack segment

What is stack memory?
Why a process stack?
Peeking at the stack
Advanced – the VM split
Summary

3: Resource Limits

Resource limits
Granularity of resource limits
Resource types
Available resource limits
Hard and soft limits
Querying and changing resource limit values
Caveats
A quick note on the prlimit utility
Using prlimit(1) – examples
API interfaces
Code examples
Permanence
Summary

4: Dynamic Memory Allocation

The glibc malloc(3) API family
The malloc(3) API
malloc(3) – some FAQs
malloc(3) – a quick summary
The free API
free – a quick summary
The calloc API
The realloc API
The realloc(3) – corner cases
The reallocarray API
Beyond the basics
The program break
Using the sbrk() API
How malloc(3) really behaves
Code example – malloc(3) and the program break
Scenario 1 – default options
Scenario 2 – showing malloc statistics
Scenario 3 – large allocations option
Where does freed memory go?
Advanced features
Demand-paging
Resident or not?
Locking memory
Limits and privileges
Locking all pages
Memory protection

Memory protection – a code example
An Aside – LSM logs, Ftrace
LSM logs
Ftrace
An experiment – running the memprot program on an ARM-32
Memory protection keys – a brief note
Using alloca to allocate automatic memory
Summary

Chapter 5: Linux Memory Issues
Common memory issues
Incorrect memory accesses
Accessing and/or using uninitialized variables
Test case 1: Uninitialized memory access
Out-of-bounds memory accesses
Test case 2
Test case 3
Test case 4
Test case 5
Test case 6
Test case 7
Use-after-free/Use-after-return bugs
Test case 8
Test case 9
Test case 10
Leakage
Test case 11
Test case 12
Test case 13
Test case 13.1
Test case 13.2
Test case 13.3
Undefined behavior
Fragmentation
Miscellaneous
Summary

6: Debugging Tools for Memory Issues
Tool types
Valgrind
Using Valgrind's Memcheck tool
Valgrind summary table
Valgrind pros and cons : a quick summary
Sanitizer tools
Sanitizer toolset
Building programs for use with ASan
Running the test cases with ASan
AddressSanitizer (ASan) summary table

AddressSanitizer pros and cons – a quick summary
Glibc mallopt
Malloc options via the environment
Some key points
Code coverage while testing
What is the modern C/C++ developer to do?
A mention of the malloc API helpers
Summary

<< **Chapter A : File I/O Essentials** ; available online >>

7: Process Credentials
The traditional Unix permissions model
Permissions at the user level
How the Unix permission model works
Determining the access category
Real and effective IDs
A puzzle – how can a regular user change their password?
The setuid and setgid special permission bits
Setting the setuid and setgid bits with chmod
Hacking attempt 1
System calls
Querying the process credentials
Code example
Sudo – how it works
What is a saved-set ID?
Setting the process credentials
Hacking attempt 2
An aside – a script to identify setuid-root and setgid installed programs
setgid example – wall
Giving up privileges
Saved-set UID – a quick demo
The setres[u|g]id(2) system calls
Important security notes
Summary

8: Process Capabilities
The modern POSIX capabilities model
Motivation
POSIX capabilities
Capabilities – some gory details
OS support
Viewing process capabilities via procfs
Thread capability sets
File capability sets
Embedding capabilities into a program binary
Capability-dumb binaries
Getcap and similar utilities

Wireshark – a case in point
Setting capabilities programmatically
Miscellaneous
How ls displays different binaries
Permission models layering
Security tips
FYI – under the hood, at the level of the Kernel
Summary

9: Process Execution

Technical requirements
Process execution
Converting a program to a process
The exec Unix axiom
Key points during an exec operation
Testing the exec axiom
Experiment 1 – on the CLI, no frills
Experiment 2 – on the CLI, again
The point of no return
Family time – the exec family APIs
The wrong way
Error handling and the exec
Passing a zero as an argument
Specifying the name of the successor
The remaining exec family APIs
The execlp API
The execle API
The execv API
Exec at the OS level
Summary table – exec family of APIs
Code example
Summary

10: Process Creation

Process creation
How fork works
Using the fork system call
Fork rule #1
Fork rule #2 – the return
Fork rule #3
Atomic execution?
Fork rule #4 – data
Fork rule #5 – racing
The process and open files
Fork rule #6 – open files
Open files and security
Malloc and the fork
COW in a nutshell

Waiting and our simpsh project
The Unix fork-exec semantic
The need to wait
Performing the wait
Defeating the race after fork
Putting it together – our simpsh project
The wait API – details
The scenarios of wait
Wait scenario #1
Wait scenario #2
Fork bombs and creating more than one child
Wait scenario #3
Variations on the wait – APIs
The waitpid(2)
The waitid (2)
The actual system call
A note on the vfork
More Unix weirdness
Orphans
Zombies
Fork rule #7
The rules of fork – a summary
Summary

11: Signaling - Part I

Why signals?
The signal mechanism in brief
Available signals
The standard or Unix signals
Handling signals
Using the sigaction system call to trap signals
Sidebar – the feature test macros
The sigaction structure
Masking signals
Signal masking with the sigprocmask API
Querying the signal mask
Sidebar – signal handling within the OS – polling not interrupts
Reentrant safety and signalling
Reentrant functions
Async-signal-safe functions
Alternate ways to be safe within a signal handler
Signal-safe atomic integers
Powerful sigaction flags
Zombies not invited
No zombies! – the classic way
No zombies! – the modern way
The SA_NOCLDSTOP flag
Interrupted system calls and how to fix them with the SA_RESTART
The once only SA_RESETHAND flag

            To defer or not? Working with SA_NODEFER
                Signal behavior when masked
                Case 1 : Default : SA_NODEFER bit cleared
                Case 2 : SA_NODEFER bit set
                Running of case 1 – SA_NODEFER bit cleared [default]
                Running of case 2 – SA_NODEFER bit set
            Using an alternate signal stack
                Implementation to handle high-volume signals with an alternate signal stack
                Case 1 – very small (100 KB) alternate signal stack
                Case 2 : A large (16 MB) alternate signal stack
            Different approaches to handling signals at high volume
        Summary

12: Signaling - Part II

Gracefully handling process crashes
    Detailing information with the SA_SIGINFO
        The siginfo_t structure
        Getting system-level details when a process crashes
            Trapping and extracting information from a crash
            Register dumping
            Finding the crash location in source code
Signaling – caveats and gotchas
    Handling errno gracefully
        What does errno do?
        The errno race
        Fixing the errno race
    Sleeping correctly
        The nanosleep system call
Real-time signals
    Differences from standard signals
        Real time signals and priority
Sending signals
    Just kill 'em
        Killing yourself with a raise
        Agent 00 – permission to kill
        Are you there?
    Signaling as IPC
        Crude IPC
        Better IPC – sending a data item
            Sidebar – LTTng
Alternative signal-handling techniques
    Synchronously waiting for signals
        Pause, please
            Waiting forever or until a signal arrives
        Synchronously blocking for signals via the sigwait* APIs
            The sigwait library API
            The sigwaitinfo and the sigtimedwait system calls
        The signalfd(2) API

Summary

13: Timers

Older interfaces
The good ol' alarm clock
Alarm API – the downer
Interval timers
A simple CLI digital clock
Obtaining the current time
Trial runs
A word on using the profiling timers
The newer POSIX (interval) timers mechanism
Typical application workflow
Creating and using a POSIX (interval) timer
The arms race – arming and disarming a POSIX timer
Querying the timer
Example code snippet showing the workflow
Figuring the overrun
POSIX interval timers – example programs
The reaction – time game
How fast is fast?
Our react game – how it works
React – trial runs
The react game – code view
The run:walk interval timer application
A few trial runs
The low – level design and code
Timer lookup via proc
A quick mention
Timers via file descriptors
A quick note on watchdog timers
Summary

14: Multithreading with Pthreads Part I - Essentials

Multithreading concepts
What exactly is a thread?
Resource sharing
Multiprocess versus multithreaded
Example 1 – creation/destruction – process/thread
The multithreading model
Example 2 – matrix multiplication – process/thread
Example 3 – kernel build
On a VM with 1 GB RAM, two CPU cores and parallelized make -j4
On a VM with 1 GB RAM, one CPU core and sequential make -j1
Motivation – why threads?
Design motivation
Taking advantage of potential parallelism
Logical separation

Overlapping CPU with I/O
Manager-worker model
IPC becoming simple(r)
Performance motivation
Creation and destruction
Automatically taking advantage of modern hardware
Resource sharing
Context switching
A brief history of threading
POSIX threads
Pthreads and Linux
Thread management – the essential pthread APIs
Thread creation
Termination
The return of the ghost
So many ways to die
How many threads is too many?
How many threads can you create?
Code example – creating any number of threads
How many threads should one create?
Thread attributes
Code example – querying the default thread attributes
Joining
The thread model join and the process model wait
Checking for life, timing out
Join or not?
Parameter passing
Passing a structure as a parameter
Thread parameters – what not to do
Thread stacks
Get and set thread stack size
Stack location
Stack guards
Summary

15: Multithreading with Pthreads Part II - Synchronization
The racing problem
Concurrency and atomicity
The pedagogical bank account example
Critical sections
Locking concepts
Is it atomic?
Dirty reads
Locking guidelines
Locking granularity
Deadlock and its avoidance
Common deadlock types
Self deadlock (relock)
The ABBA deadlock

Avoiding deadlock

Using the pthread APIs for synchronization

The mutex lock

Seeing the race

Mutex attributes

Mutex types

The robust mutex attribute

IPC, threads, and the process-shared mutex

Priority inversion, watchdogs, and Mars

Priority inversion

Watchdog timer in brief

The Mars Pathfinder mission in brief

Priority inheritance – avoiding priority inversion

Summary of mutex attribute usage

Mutex locking – additional variants

Timing out on a mutex lock attempt

Busy-waiting (non-blocking variant) for the lock

The reader-writer mutex lock

The spinlock variant

A few more mutex usage guidelines

Is the mutex locked?

Condition variables

No CV – the naive approach

Using the condition variable

A simple CV usage demo application

CV broadcast wakeup

Summary

16: Multithreading with Pthreads Part III

Thread safety

Making code thread-safe

Reentrant-safe versus thread-safe

Summary table – approaches to making functions thread-safe

Thread safety via mutex locks

Thread safety via function refactoring

The standard C library and thread safety

List of APIs not required to be thread-safe

Refactoring glibc APIs from foo to foo_r

Some glibc foo and foo_r APIs

Thread safety via TLS

Thread safety via TSD

Thread cancelation and cleanup

Canceling a thread

The thread cancelation framework

The cancelability state

The cancelability type

Canceling a thread – a code example

Cleaning up at thread exit

Thread cleanup – code example

Threads and signaling
The issue
The POSIX solution to handling signals on MT
Code example – handling signals in an MT app
Threads vs processes – look again
The multiprocess vs the multithreading model – pros of the MT model 689
The multiprocess vs the multithreading model – cons of the MT model 690
Pthreads – a few random tips and FAQs
Pthreads – some FAQs
Debugging multithreaded (pthreads) applications with GDB
Summary

17: CPU Scheduling on Linux
The Linux OS and the POSIX scheduling model
The Linux process state machine
The sleep states
What is real time?
Types of real time
Scheduling policies
Peeking at the scheduling policy and priority
The nice value
CPU affinity
Exploiting Linux's soft real-time capabilities
Scheduling policy and priority APIs
Code example – setting a thread scheduling policy and priority
Soft real-time – additional considerations
RTL – Linux as an RTOS
Summary

18: Advanced File I/O
I/O performance recommendations
The kernel page cache
Giving hints to the kernel on file I/O patterns
Via the posix_fadvise(2) API
Via the readahead(2) API
MT app file I/O with the pread, pwrite APIs
Scatter – gather I/O
Discontiguous data file – traditional approach
Discontiguous data file – the SG – I/O approach
SG – I/O variations
File I/O via memory mapping
The Linux I/O code path in brief
Memory mapping a file for I/O
File and anonymous mappings
The mmap advantage
Code example
Memory mapping – additional points
DIO and AIO

Direct I/O (DIO)
Asynchronous I/O (AIO)
I/O technologies – a quick comparison
Multiplexing or async blocking I/O – a quick note
I/O – miscellaneous
Linux's inotify framework
I/O schedulers
Ensuring sufficient disk space
Utilities for I/O monitoring, analysis, and bandwidth control
Summary

<< **Chapter B : Daemon Processes** ; available online >>

19: Troubleshooting and Best Practices
Troubleshooting tools
perf
Tracing tools
The Linux proc filesystem
Best practices
The empirical approach
Software engineering wisdom in a nutshell
Programming
A programmer's checklist – seven rules
Better testing
Using the Linux kernel's control groups
Summary.

Authors

Kaiwan Billimoria

Kaiwan Billimoria taught himself BASIC programming on his Dad's office IBM PC when he was in the 9th grade (back in 1983). The urge to learn, hack and master at the level of the "bare-metal" was born there!

After completing his B.E. in Information Science & Tech, he went to work: helping build an antivirus product using assembly and C on a DOS platform. He thought DOS was awesome, until one fateful day he "discovered" the joys of Unix! (from none other than Richard Steven's iconic original "UNIX Network Programming" book and working on 'C' applications on SCO Unix). Many years later, working on the creative side (multimedia, web design, HTML), he "re-discovered" Free Unix on the PC - in the form of Linux of course.

Kaiwan has plumbed the depths of the Linux system programming stack – right from mastering bash scripting, to system programming in userspace with 'C', to deep OS-level kernel internals hacking and debugging. Along the way, he has worked on several commercial and opensource products and projects. His (opensource) contributions include drivers for the mainline Linux OS, and several small self-founded projects hosted on GitHub. He is currently working on and looking to actively contribute in the Linux OS kernel hardening and security space.

His passion for Linux and actively programming on it by working on industry and opensource projects and products, feeds well into his other passion – for two decades now, Kaiwan has been teaching engineers the intracacies of the Linux OS, and how to design and implement code (for robustness and security).

It doesn't hurt that he is an ultra-marathoner too (several marathons, a few 50k's and a memorable 100 km run).