Table of Contents

Preface 1

Section 1: The Basics

Chapter 1: Kernel Workspace Setup 13
- **Technical requirements** 14
- **Running Linux as a guest VM** 15
  - Installing a 64-bit Linux guest 15
    - Turn on your x86 system's virtualization extension support 16
    - Allocate sufficient space to the disk 18
    - Install the Oracle VirtualBox Guest Additions 18
  - Experimenting with the Raspberry Pi 21
- **Setting up the software – distribution and packages** 24
  - Installing software packages 26
    - Installing the Oracle VirtualBox guest additions 27
    - Installing required software packages 27
    - Installing a cross toolchain and QEMU 28
    - Installing a cross compiler 29
  - Important installation notes 30
- **Additional useful projects** 31
  - Using the Linux man pages 31
    - The tldr variant 32
  - Locating and using the Linux kernel documentation 33
    - Generating the kernel documentation from source 35
  - Static analysis tools for the Linux kernel 35
  - Linux Trace Toolkit next generation 36
  - The procmap utility 37
  - Simple Embedded ARM Linux System FOSS project 39
  - Modern tracing and performance analysis with [e]BPF 39
  - The LDV - Linux Driver Verification - project 40
- **Summary** 42
- **Questions** 42
- **Further reading** 42

Chapter 2: Building the 5.x Linux Kernel from Source - Part 1 43
- **Technical requirements** 44
- **Preliminaries for the kernel build** 44
  - Kernel release nomenclature 45
  - Kernel development workflow – the basics 46
  - Types of kernel source trees 49
- **Steps to build the kernel from source** 52

Step 1 – obtaining a Linux kernel source tree 53
Downloading a specific kernel tree 54
Cloning a Git tree 55
Step 2 – extracting the kernel source tree 57
A brief tour of the kernel source tree 58
Step 3 – configuring the Linux kernel 63
Understanding the kbuild build system 64
Arriving at a default configuration 65
Obtaining a good starting point for kernel configuration 66
Kernel config for typical embedded Linux systems 67
Kernel config using distribution config as a starting point 68
Tuned kernel config via the localmodconfig approach 69
Getting started with the localmodconfig approach 71
Tuning our kernel configuration via the make menuconfig UI 74
Sample usage of the make menuconfig UI 77
More on kbuild 85
Looking up the differences in configuration 86
Customizing the kernel menu – adding our own menu item 88
The Kconfig* files 89
Creating a new menu item in the Kconfig file 90
A few details on the Kconfig language 93
Summary 96
Questions 96
Further reading 96
Chapter 3: Building the 5.x Linux Kernel from Source - Part 2 97
Technical requirements 98
Step 4 – building the kernel image and modules 98
Step 5 – installing the kernel modules 104
Locating the kernel modules within the kernel source 104
Getting the kernel modules installed 106
Step 6 – generating the initramfs image and bootloader setup 108
Generating the initramfs image on Fedora 30 and above 109
Generating the initramfs image – under the hood 110
Understanding the initramfs framework 112
Why the initramfs framework? 112
Understanding the basics of the boot process on the x86 115
More on the initramfs framework 116
Step 7 – customizing the GRUB bootloader 119
Customizing GRUB – the basics 120
Selecting the default kernel to boot into 121
Booting our VM via the GNU GRUB bootloader 123
Experimenting with the GRUB prompt 125
Verifying our new kernel's configuration 127
Kernel build for the Raspberry Pi 129

Step 1 – cloning the kernel source tree 130
Step 2 – installing a cross-toolchain 131
First method – package install via apt 132
Second method – installation via the source repo 132
Step 3 – configuring and building the kernel 133
Miscellaneous tips on the kernel build 136
Minimum version requirements 137
Building a kernel for another site 137
Watching the kernel build run 138
A shortcut shell syntax to the build procedure 139
Dealing with compiler switch issues 140
Dealing with missing OpenSSL development headers 140
Summary 142
Questions 142
Further reading 142
Chapter 4: Writing Your First Kernel Module - LKMs Part 1 145
Technical requirements 146
Understanding kernel architecture – part 1 147
User space and kernel space 147
Library and system call APIs 148
Kernel space components 149
Exploring LKMs 152
The LKM framework 152
Kernel modules within the kernel source tree 154
Writing our very first kernel module 157
Introducing our Hello, world LKM C code 157
Breaking it down 158
Kernel headers 159
Module macros 160
Entry and exit points 160
Return values 161
The 0/-E return convention 161
The ERR_PTR and PTR_ERR macros 163
The __init and __exit keywords 164
Common operations on kernel modules 165
Building the kernel module 165
Running the kernel module 167
A quick first look at the kernel printk() 168
Listing the live kernel modules 171
Unloading the module from kernel memory 172
Our lkm convenience script 173
Understanding kernel logging and printk 175
Using the kernel memory ring buffer 176
Kernel logging and systemd's journalctl 178
Using printk log levels 180

The pr_<foo> convenience macros 182
Wiring to the console 183
Writing output to the Raspberry Pi console 185
Enabling the pr_debug() kernel messages 188
Rate limiting the printk instances 190
Generating kernel messages from the user space 193
Standardizing printk output via the pr_fmt macro 194
Portability and the printk format specifiers 196
Understanding the basics of a kernel module Makefile 197
Summary 200
Questions 200
Further reading 200
Chapter 5: Writing Your First Kernel Module - LKMs Part 2 201
Technical requirements 202
A "better" Makefile template for your kernel modules 202
Configuring a "debug" kernel 204
Cross-compiling a kernel module 206
Setting up the system for cross-compilation 207
Attempt 1 – setting the "special" environment variables 208
Attempt 2 – pointing the Makefile to the correct kernel source tree for the target 211
Attempt 3 – cross-compiling our kernel module 213
Attempt 4 – cross-compiling our kernel module 217
Gathering minimal system information 219
Being a bit more security-aware 222
Licensing kernel modules 224
Emulating "library-like" features for kernel modules 226
Performing library emulation via multiple source files 226
Understanding function and variable scope in a kernel module 227
Understanding module stacking 231
Trying out module stacking 234
Passing parameters to a kernel module 241
Declaring and using module parameters 241
Getting/setting module parameters after insertion 244
Module parameter data types and validation 247
Validating kernel module parameters 247
Overriding the module parameter's name 248
Hardware-related kernel parameters 249
Floating point not allowed in the kernel 250
Auto-loading modules on system boot 252
Module auto-loading – additional details 256
Kernel modules and security – an overview 258
Proc filesystem tunables affecting the system log 259
The cryptographic signing of kernel modules 261
Disabling kernel modules altogether 263

Coding style guidelines for kernel developers 264
Contributing to the mainline kernel 265
Getting started with contributing to the kernel 265
Summary 266
Questions 267
Further reading 267

Section 2: Understanding and Working with the Kernel

Chapter 6: Kernel Internals Essentials - Processes and Threads 271
Technical requirements 272
Understanding process and interrupt contexts 272
Understanding the basics of the process VAS 274
Organizing processes, threads, and their stacks – user and kernel space 277
User space organization 280
Kernel space organization 282
Summarizing the current situation 284
Viewing the user and kernel stacks 285
Traditional approach to viewing the stacks 286
Viewing the kernel space stack of a given thread or process 286
Viewing the user space stack of a given thread or process 287
[e]BPF – the modern approach to viewing both stacks 288
The 10,000-foot view of the process VAS 292
Understanding and accessing the kernel task structure 294
Looking into the task structure 296
Accessing the task structure with current 298
Determining the context 299
Working with the task structure via current 301
Built-in kernel helper methods and optimizations 302
Trying out the kernel module to print process context info 304
Seeing that the Linux OS is monolithic 305
Coding for security with printk 305
Iterating over the kernel's task lists 307
Iterating over the task list I – displaying all processes 307
Iterating over the task list II – displaying all threads 308
Differentiating between the process and thread – the TGID and the PID 310
Iterating over the task list III – the code 312
Summary 315
Questions 316
Further reading 316
Chapter 7: Memory Management Internals - Essentials 317
Technical requirements 318
Understanding the VM split 318

Looking under the hood – the Hello, world C program 319
Going beyond the printf() API 321
VM split on 64-bit Linux systems 324
Virtual addressing and address translation 324
The process VAS – the full view 330
Examining the process VAS 331
Examining the user VAS in detail 332
Directly viewing the process memory map using procfs 333
Interpreting the /proc/PID/maps output 333
The vsyscall page 336
Frontends to view the process memory map 336
The procmap process VAS visualization utility 337
Understanding VMA basics 342
Examining the kernel segment 344
High memory on 32-bit systems 347
Writing a kernel module to show information about the kernel segment 348
Viewing the kernel segment on a Raspberry Pi via dmesg 348
Macros and variables describing the kernel segment layout 350
Trying it out – viewing kernel segment details 353
The kernel VAS via procmap 357
Trying it out – the user segment 362
The null trap page 364
Viewing kernel documentation on the memory layout 365
Randomizing the memory layout – KASLR 366
User-mode ASLR 367
KASLR 368
Querying/setting KASLR status with a script 368
Physical memory 372
Physical RAM organization 372
Nodes 372
Zones 375
Direct-mapped RAM and address translation 377
Summary 382
Questions 382
Further reading 382
Chapter 8: Kernel Memory Allocation for Module Authors - Part 1 383
Technical requirements 384
Introducing kernel memory allocators 384
Understanding and using the kernel page allocator (or BSA) 386
The fundamental workings of the page allocator 387
Freelist organization 387
The workings of the page allocator 390
Working through a few scenarios 392
The simplest case 392
A more complex case 392
The downfall case 392
Page allocator internals – a few more details 393

Learning how to use the page allocator APIs 395
Dealing with the GFP flags 397
Freeing pages with the page allocator 399
Writing a kernel module to demo using the page allocator APIs 401
Deploying our lowlevel_mem_lkm kernel module 407
The page allocator and internal fragmentation 411
The exact page allocator APIs 411
The GFP flags – digging deeper 413
Never sleep in interrupt or atomic contexts 413
Understanding and using the kernel slab allocator 415
The object caching idea 416
Learning how to use the slab allocator APIs 419
Allocating slab memory 419
Freeing slab memory 421
Data structures – a few design tips 423
The actual slab caches in use for kmalloc 424
Writing a kernel module to use the basic slab APIs 426
Size limitations of the kmalloc API 429
Testing the limits – memory allocation with a single call 430
Checking via the /proc/buddyinfo pseudo-file 433
Slab allocator – a few additional details 435
Using the kernel's resource-managed memory allocation APIs 435
Additional slab helper APIs 436
Control groups and memory 437
Caveats when using the slab allocator 438
Background details and conclusions 438
Testing slab allocation with ksize() – case 1 439
Testing slab allocation with ksize() – case 2 440
Interpreting the output from case 2 442
Graphing it 443
Slab layer implementations within the kernel 444
Summary 445
Questions 445
Further reading 445
Chapter 9: Kernel Memory Allocation for Module Authors - Part 2 447
Technical requirements 447
Creating a custom slab cache 448
Creating and using a custom slab cache within a kernel module 448
Creating a custom slab cache 449
Using the new slab cache's memory 452
Destroying the custom cache 453
Custom slab – a demo kernel module 453
Understanding slab shrinkers 458
The slab allocator – pros and cons – a summation 459
Debugging at the slab layer 460
Debugging through slab poisoning 460

Trying it out – triggering a UAF bug 462
SLUB debug options at boot and runtime 465
Understanding and using the kernel vmalloc() API 466
Learning to use the vmalloc family of APIs 467
A brief note on memory allocations and demand paging 471
Friends of vmalloc() 473
Specifying the memory protections 477
Testing it – a quick Proof of Concept 478
Why make memory read-only? 480
The kmalloc() and vmalloc() APIs – a quick comparison 480
Memory allocation in the kernel – which APIs to use when 481
Visualizing the kernel memory allocation API set 482
Selecting an appropriate API for kernel memory allocation 483
A word on DMA and CMA 486
Stayin' alive – the OOM killer 487
Reclaiming memory – a kernel housekeeping task and OOM 487
Deliberately invoking the OOM killer 488
Invoking the OOM killer via Magic SysRq 489
Invoking the OOM killer with a crazy allocator program 489
Understanding the rationale behind the OOM killer 491
Case 1 – vm.overcommit set to 2, overcommit turned off 492
Case 2 – vm.overcommit set to 0, overcommit on, the default 493
Demand paging and OOM 494
Understanding the OOM score 498
Summary 499
Questions 500
Further reading 500
Chapter 10: The CPU Scheduler - Part 1 501
Technical requirements 502
Learning about the CPU scheduling internals – part 1 – essential background 502
What is the KSE on Linux? 502
The POSIX scheduling policies 503
Visualizing the flow 506
Using perf to visualize the flow 506
Visualizing the flow via alternate (CLI) approaches 510
Learning about the CPU scheduling internals – part 2 511
Understanding modular scheduling classes 511
Asking the scheduling class 517
A word on CFS and the vruntime value 519
Threads – which scheduling policy and priority 521
Learning about the CPU scheduling internals – part 3 525
Who runs the scheduler code? 525
When does the scheduler run? 525
The timer interrupt part 526

The process context part 527
Preemptible kernel 528
CPU scheduler entry points 530
The context switch 532
Summary 533
Questions 533
Further reading 533
Chapter 11: The CPU Scheduler - Part 2 535
Technical requirements 535
Visualizing the flow with LTTng and trace-cmd 536
Visualization with LTTng and Trace Compass 536
Recording a kernel tracing session with LTTng 537
Reporting with a GUI – Trace Compass 538
Visualizing with trace-cmd 539
Recording a sample session with trace-cmd record 540
Reporting and interpretation with trace-cmd report (CLI) 542
Reporting and interpretation with a GUI frontend 547
Understanding, querying, and setting the CPU affinity mask 548
Querying and setting a thread's CPU affinity mask 550
Using taskset(1) to perform CPU affinity 554
Setting the CPU affinity mask on a kernel thread 554
Querying and setting a thread's scheduling policy and priority 555
Within the kernel – on a kernel thread 556
CPU bandwidth control with cgroups 557
Looking up cgroups v2 on a Linux system 559
Trying it out – a cgroups v2 CPU controller 561
Converting mainline Linux into an RTOS 566
Building RTL for the mainline 5.x kernel (on x86_64) 568
Obtaining the RTL patches 568
Applying the RTL patch 571
Configuring and building the RTL kernel 571
Mainline and RTL – technical differences summarized 575
Latency and its measurement 576
Measuring scheduling latency with cyclictest 578
Getting and applying the RTL patchset 579
Installing cyclictest (and other required packages) on the device 582
Running the test cases 583
Viewing the results 585
Measuring scheduler latency via modern BPF tools 588
Summary 589
Questions 590
Further reading 590

Section 3: Delving Deeper

Chapter 12: Kernel Synchronization - Part 1 593

Critical sections, exclusive execution, and atomicity 594
What is a critical section? 594
A classic case – the global i ++ 597
Concepts – the lock 600
A summary of key points 603
Concurrency concerns within the Linux kernel 604
Multicore SMP systems and data races 605
Preemptible kernels, blocking I/O, and data races 606
Hardware interrupts and data races 607
Locking guidelines and deadlocks 608
Mutex or spinlock? Which to use when 611
Determining which lock to use – in theory 613
Determining which lock to use – in practice 614
Using the mutex lock 615
Initializing the mutex lock 616
Correctly using the mutex lock 617
Mutex lock and unlock APIs and their usage 618
Mutex lock – via [un]interruptible sleep? 620
Mutex locking – an example driver 621
The mutex lock – a few remaining points 626
Mutex lock API variants 626
The mutex trylock variant 626
The mutex interruptible and killable variants 628
The mutex io variant 629
The semaphore and the mutex 629
Priority inversion and the RT-mutex 630
Internal design 631
Using the spinlock 632
Spinlock – simple usage 632
Spinlock – an example driver 634
Test – sleep in an atomic context 636
Testing on a 5.4 debug kernel 637
Testing on a 5.4 non-debug distro kernel 642
Locking and interrupts 645
Using spinlocks – a quick summary 652
Summary 653
Questions 653
Further reading 653

Chapter 13: Kernel Synchronization - Part 2 655
Using the atomic_t and refcount_t interfaces 656
The newer refcount_t versus older atomic_t interfaces 657
The simpler atomic_t and refcount_t interfaces 659
Examples of using refcount_t within the kernel code base 661
64-bit atomic integer operators 663
Using the RMW atomic operators 665

RMW atomic operations – operating on device registers 665
Using the RMW bitwise operators 668
Using bitwise atomic operators – an example 670
Efficiently searching a bitmask 673
Using the reader-writer spinlock 673
Reader-writer spinlock interfaces 674
A word of caution 676
The reader-writer semaphore 677
Cache effects and false sharing 678
Lock-free programming with per-CPU variables 680
Per-CPU variables 681
Working with per-CPU 682
Allocating, initialization, and freeing per-CPU variables 682
Performing I/O (reads and writes) on per-CPU variables 683
Per-CPU – an example kernel module 685
Per-CPU usage within the kernel 689
Lock debugging within the kernel 691
Configuring a debug kernel for lock debugging 692
The lock validator lockdep – catching locking issues early 694
Examples – catching deadlock bugs with lockdep 697
Example 1 – catching a self deadlock bug with lockdep 697
Fixing it 701
Example 2 – catching an AB-BA deadlock with lockdep 702
lockdep – annotations and issues 707
lockdep annotations 707
lockdep issues 708
Lock statistics 709
Viewing lock stats 709
Memory barriers – an introduction 711
An example of using memory barriers in a device driver 712
Summary 714
Questions 715
Further reading 715
717
Other Books You May Enjoy 719
Index 723