Step 1 â€“ obtaining a Linux kernel source tree
Downloading a specific kernel tree
Cloning a Git tree
Step 2 â€“ extracting the kernel source tree
A brief tour of the kernel source tree
Step 3 â€“ configuring the Linux kernel
Understanding the kbuild build system
Arriving at a default configuration
Obtaining a good starting point for kernel configuration
Kernel config for typical embedded Linux systems
Kernel config using distribution config as a starting point
Tuned kernel config via the localmodconfig approach
Getting started with the localmodconfig approach
Tuning our kernel configuration via the make menuconfig UI
Sample usage of the make menuconfig UI
More on kbuild
Looking up the differences in configuration
Customizing the kernel menu â€“ adding our own menu item
The Kconfig files*
Creating a new menu item in the Kconfig file
A few details on the Kconfig language
Summary
Questions
Further reading
Building the 5.x Linux Kernel from Source - Part 2
Technical requirements
Step 4 â€“ building the kernel image and modules
Step 5 â€“ installing the kernel modules
Locating the kernel modules within the kernel source
Getting the kernel modules installed
Step 6 â€“ generating the initramfs image and bootloader setup
Generating the initramfs image on Fedora 30 and above
Generating the initramfs image â€“ under the hood
Understanding the initramfs framework
Why the initramfs framework?
Understanding the basics of the boot process on the x86
More on the initramfs framework
Step 7 â€“ customizing the GRUB bootloader
Customizing GRUB â€“ the basics
Selecting the default kernel to boot into
Booting our VM via the GNU GRUB bootloader
Experimenting with the GRUB prompt
Verifying our new kernel's configuration
Kernel build for the Raspberry Pi
Step 1 â€“ cloning the kernel source tree
Step 2 â€“ installing a cross-toolchain

First method â€“ package install via apt
Second method â€“ installation via the source repo
Step 3 â€“ configuring and building the kernel
Miscellaneous tips on the kernel build
Minimum version requirements
Building a kernel for another site
Watching the kernel build run
A shortcut shell syntax to the build procedureÂ
Dealing withÂ compiler switch issues
Dealing withÂ missingÂ OpenSSLÂ development headers
Summary
Questions
Further reading
Writing Your First Kernel Module - LKMs Part 1
Technical requirements
Understanding kernel architecture â€“ part 1
User space and kernel space
Library and system call APIs
Kernel space components
Exploring LKMs
The LKM framework
Kernel modules within the kernel source tree
Writing our very first kernel module
Introducing our Hello, world LKM C code
Breaking it down
Kernel headers
Module macros
Entry and exit points
Return values
The 0/-E return convention
The ERR_PTR and PTR_ERR macros
The __init and __exit keywords
Common operations on kernel modules
Building the kernel module
Running the kernel module
A quick first look at the kernel printk()
Listing the live kernel modules
Unloading the module from kernel memory
Our lkm convenience script
Understanding kernel logging and printk
Using the kernel memory ring buffer
Kernel logging and systemd's journalctl
Using printk log levels
The pr_ convenience macros
Wiring to the console
Writing output to the Raspberry Pi console

Enabling the pr_debug() kernel messages
Rate limiting the printk instances
Generating kernel messages from the user space
Standardizing printk output via the pr_fmt macro
Portability and the printk format specifiers
Understanding the basics of a kernel module Makefile
SummaryÂ
Questions
Further reading
Writing Your First Kernel Module - LKMs Part 2
Technical requirements
A "better" Makefile template for your kernel modules
Configuring a "debug" kernel
Cross-compiling a kernel module
Setting up the system for cross-compilation
Attempt 1 â€“ setting the "special" environment variables
Attempt 2 â€“ pointing the MakefileÂ to the correct kernel source tree for the target
Attempt 3 â€“ cross-compiling our kernel module
Attempt 4Â â€“ cross-compiling our kernel module
Gathering minimal system information
Being a bit more security-aware
Licensing kernel modules
Emulating "library-like" features for kernel modules
Performing library emulation via multiple source files
Understanding function and variable scope in a kernel module
Understanding module stacking
Trying out module stacking
Passing parameters to a kernel module
Declaring and using module parameters
Getting/setting module parameters after insertion
Module parameter data types and validation
Validating kernel module parameters
Overriding the module parameter's name
Hardware-related kernel parameters
Floating point not allowed in the kernel
Auto-loading modules on system boot
Module auto-loading â€“ additional details
Kernel modules and security â€“ an overview
Proc filesystem tunables affecting the system log
The cryptographic signing of kernel modules
Disabling kernel modules altogether
Coding style guidelines for kernel developers
Contributing to the mainline kernel
Getting started with contributing to the kernel
Summary
Questions

Further reading
Section 2: Understanding and Working with the Kernel
Kernel Internals Essentials - Processes and Threads
Technical requirements
Understanding process and interrupt contexts
Understanding the basics of the process VAS
Organizing processes, threads, and their stacks â€“ user and kernel space
User space organization
Kernel space organization
Summarizing the current situation
Viewing the user and kernel stacks
Traditional approach to viewing the stacks
Viewing the kernel space stack of a given thread or process
Viewing the user space stack of a given thread or process
[e]BPF â€“ the modern approach to viewing both stacks
The 10,000-foot view of the process VAS
Understanding and accessing the kernel task structure
Looking into the task structure
Accessing the task structure with current
Determining the context
Working with the task structure via current
Built-in kernel helper methods and optimizations
Trying out the kernel module to print process context info
Seeing that the Linux OS is monolithic
Coding for security with printk
Iterating over the kernel's task lists
Iterating over the task list I â€“ displaying all processes
Iterating over the task list II â€“ displaying all threads
Differentiating between the process and thread â€“ the TGID and the PID
Iterating over the task list III â€“ the code
Summary
Questions
Further reading
Memory Management Internals - Essentials
Technical requirements
Understanding the VM split
Looking under the hood â€“ the Hello, world C program
Going beyond the printf() API
VM split on 64-bit Linux systems
Virtual addressing and address translation
The process VAS â€“ the full view
Examining the process VAS
Examining the user VAS in detail
Directly viewing the process memory map using procfs
Interpreting the /proc/PID/maps output
The vsyscall page

Frontends to view the process memory map
The procmap process VAS visualization utility
Understanding VMA basics
Examining the kernel segment
High memory on 32-bit systems
Writing a kernel module to show information about the kernel segment
Viewing the kernel segment on a Raspberry Pi via dmesg
Macros and variables describing the kernel segment layout
Trying it out â€“ viewing kernel segment details
The kernel VAS via procmap
Trying it out â€“ the user segment
The null trap page
Viewing kernel documentation on the memory layout
Randomizing the memory layout â€“ KASLR
User-mode ASLR
KASLR
Querying/setting KASLR status with a script
Physical memory
Physical RAM organization
Nodes
Zones
Direct-mapped RAM and address translation
Summary
Questions
Further reading
Kernel Memory Allocation for Module Authors - Part 1
Technical requirements
Introducing kernel memory allocators
Understanding and using the kernel page allocator (or BSA)
The fundamental workings of the page allocator
Freelist organization
The workings of the page allocator
Working through a few scenarios
The simplest case
A more complex case
The downfall case
Page allocator internals â€“ a few more details
Learning how to use the page allocator APIs
Dealing with the GFP flags
Freeing pages with the page allocator
Writing a kernel module to demo using the page allocator APIs
Deploying our lowlevel_mem_lkm kernel module
The page allocator and internal fragmentation
The exact page allocator APIs
The GFP flags â€“ digging deeper
Never sleep in interrupt or atomic contexts

Understanding and using the kernel slab allocator
The object caching idea
Learning how to use the slab allocator APIs
Allocating slab memory
Freeing slab memory
Data structures â€“ a few design tips
The actual slab caches in use for kmalloc
Writing aÂ kernel module to use the basic slab APIs
Size limitations of the kmalloc API
Testing the limits â€“ memory allocation with a single call
Checking via the /proc/buddyinfo pseudo-file
Slab allocator â€“ a few additional details
Using the kernel's resource-managed memory allocation APIs
Additional slab helper APIs
Control groups and memory
Caveats when using the slab allocator
Background details and conclusions
Testing slab allocation with ksize() â€“ case 1
Testing slab allocation with ksize() â€“ case 2
Interpreting the output from case 2
Graphing it
Slab layer implementations within the kernel
Summary
Questions
Further reading
Kernel Memory Allocation for Module Authors - Part 2
Technical requirements
Creating a custom slab cache
Creating and using a custom slab cache within a kernel module
Creating a custom slab cache
Using the new slab cache's memory
Destroying the custom cache
Custom slab â€“ a demo kernel module
Understanding slab shrinkers
The slab allocator â€“ pros and cons â€“ a summation
Debugging at the slab layer
Debugging through slab poisoning
Trying it out â€“ triggering a UAF bug
SLUB debug options at boot and runtime
Understanding and using the kernel vmalloc() API
Learning to use the vmalloc family of APIs
A brief note on memory allocations and demand paging
Friends of vmalloc()
Specifying the memory protections
Testing it â€“ a quick Proof of Concept
Why make memory read-only?

The kmalloc() and vmalloc() APIs â€“ a quick comparison
Memory allocation in the kernel â€“ which APIs to use when
Visualizing the kernel memory allocation API set
Selecting an appropriate API for kernel memory allocation
A word on DMA and CMA
Stayin' alive â€“ the OOM killer
Reclaiming memory â€“ a kernel housekeeping task and OOM
Deliberately invoking the OOM killer
Invoking the OOM killer via Magic SysRq
Invoking the OOM killer with a crazy allocator program
Understanding the rationale behind the OOM killer
Case 1 â€“ vm.overcommit set to 2, overcommit turned off
Case 2 â€“ vm.overcommit set to 0, overcommit on, the default
Demand paging and OOM
Understanding the OOM score
Summary
Questions
Further reading
The CPU Scheduler - Part 1
Technical requirements
Learning about the CPU scheduling internals â€“ part 1 â€“ essential background
What is the KSE on Linux?
The POSIX scheduling policies
Visualizing the flow
Using perf to visualize the flow
Visualizing the flow via alternate (CLI) approaches
Learning about the CPU scheduling internals â€“ part 2
Understanding modular scheduling classes
Asking the scheduling class
A word on CFS and the vruntime value
Threads â€“ which scheduling policy and priority
Learning about the CPU scheduling internals â€“ part 3
Who runs the scheduler code?
When does the scheduler run?
The timer interrupt part
The process context part
Preemptible kernel
CPU scheduler entry points
The context switch
Summary
Questions
Further reading
The CPU Scheduler - Part 2
Technical requirements
Visualizing the flow with LTTng and trace-cmd
Visualization with LTTng and Trace Compass

Recording a kernel tracing session with LTTng
Reporting with a GUI â€“ Trace Compass
Visualizing with trace-cmd
Recording a sample session with trace-cmd record
Reporting and interpretation with trace-cmd report (CLI)
Reporting and interpretation with a GUI frontend
Understanding,Â querying, and setting the CPU affinity mask
Querying and setting a thread's CPU affinity mask
Using taskset(1) to perform CPU affinity
Setting the CPU affinity mask on a kernel thread
Querying and setting a threadâ€™s scheduling policy and priority
Within the kernel â€“ on a kernel thread
CPU bandwidth control with cgroups
Looking up cgroups v2 on a Linux system
Trying it out â€“ a cgroups v2 CPU controller
Converting mainline Linux into an RTOS
Building RTL for the mainline 5.x kernel (on x86_64)
Obtaining the RTL patches
Applying the RTL patch
Configuring and building the RTL kernel
Mainline and RTL â€“ technical differences summarized
Latency and its measurement
Measuring scheduling latency with cyclictest
Getting and applying the RTL patchset
Installing cyclictest (and other required packages) on the device
Running the test cases
Viewing the results
Measuring scheduler latency via modern BPF tools
Summary
Questions
Further reading
Section 3: Delving Deeper
Kernel Synchronization - Part 1
Critical sections, exclusive execution, and atomicity
What is a critical section?
A classic case â€“ the global i ++
Concepts â€“ the lock
A summary of key points
Concurrency concerns withinÂ the Linux kernel
Multicore SMP systems and data races
Preemptible kernels, blocking I/O, and data races
Hardware interrupts and data races
Locking guidelines and deadlocks
Mutex or spinlock? Which to use when
Determining which lock to use â€“ in theory
Determining which lock to use â€“ in practice

Using the mutex lock
Initializing the mutex lock
Correctly using the mutex lock
Mutex lock and unlock APIs and their usage
Mutex lock â€“ Â via [un]interruptible sleep?
Mutex locking â€“ an example driver
The mutex lock â€“ a few remaining points
Mutex lock API variants
The mutex trylock variant
The mutex interruptible and killable variants
The mutex io variant
The semaphore and the mutex
Priority inversion and the RT-mutex
Internal design
Using the spinlock
Spinlock â€“ simple usage
Spinlock â€“ an example driver
Test â€“ sleep in an atomic context
Testing on a 5.4 debug kernel
Testing on a 5.4 non-debug distro kernel
Locking and interrupts
Using spinlocksÂ â€“ a quick summary
Summary
Questions
Further reading
Kernel Synchronization - Part 2
Using the atomic_t and refcount_t interfaces
The newer refcount_t versus older atomic_t interfaces
The simpler atomic_t and refcount_t interfaces
Examples of using refcount_t within the kernel code base
64-bit atomic integer operators
Using the RMW atomic operators
RMW atomic operations â€“ operating on device registers
Using the RMW bitwise operators
Using bitwise atomic operators â€“ an example
Efficiently searching a bitmask
Using the reader-writer spinlock
Reader-writer spinlock interfaces
A word of caution
The reader-writer semaphore
Cache effects and false sharing
Lock-free programming with per-CPU variables
Per-CPU variables
Working with per-CPU
Allocating, initialization, and freeing per-CPU variables
Performing I/O (reads and writes) on per-CPU variables