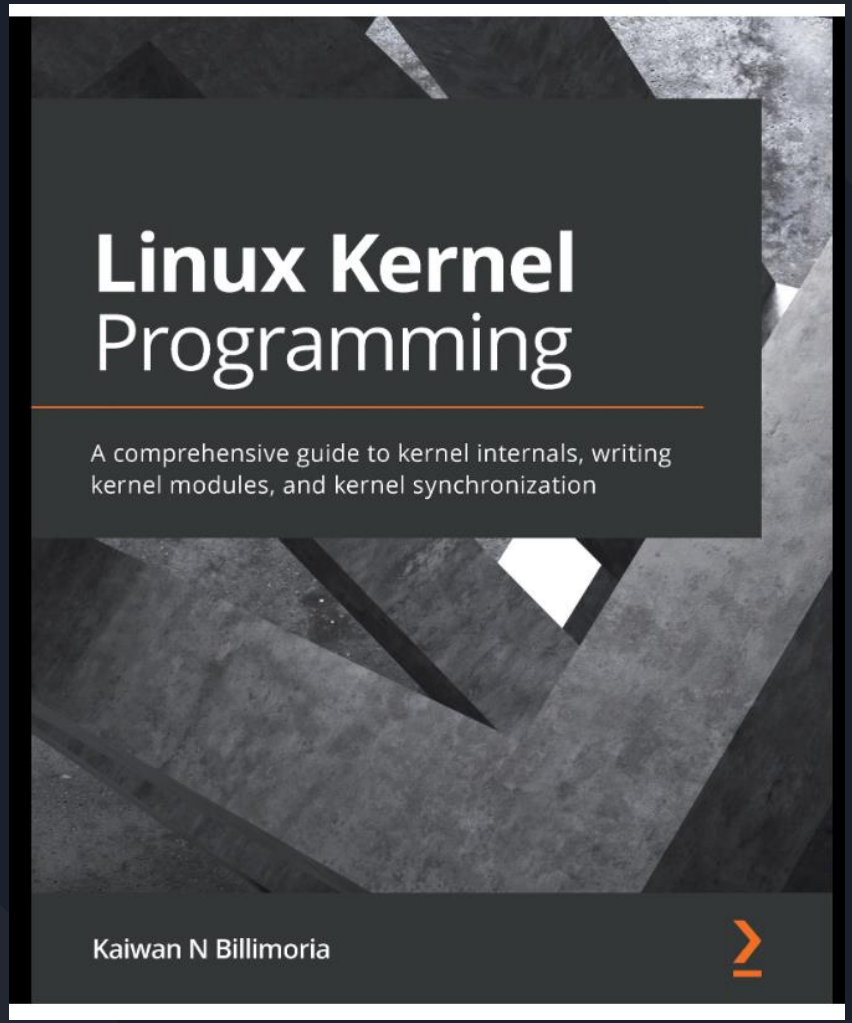
Practical Linux Kernel Development

Presented by Timothy Finnegan

About Me

- B.S. in Computer Science and Engineering - UNR Spring 2021
- M.S. in Computer Science and Engineering - UNR est. Spring 2023
- One of three primary developers on the university's Nevada Cyber Range cybersecurity education platform.
- Research deals with Android static malware analysis through machine learning.


Linux Kernel
Programming
A comprehensive guide to kernel internals, writing kernel modules, and kernel synchronization
Kaiwan N Billimoria

Lesson Outline

- Explore what interacting directly with the Linux kernel looks like
- Describe Linux's "Loadable Kernel Module" (LKM) system.
  - How do you write code that is intended to run within the Kernel, rather than user space?
- What are the Kernel resources available to you from within an LKM?
- A brief outline of some core Kernel Module API functions and functionality.

Major Kernel Subsystems

1. The Core Kernel -- Various topics from this class related to multiprogramming. CPU scheduling, threads, processes, interrupts, etc.
2. Memory Management -- Manages Virtual Address Spaces (VASes) for the Kernel and userspace processes
3. The Virtual Filesystem Switch (VFS) -- The main kernel supports a variety of filesystems (ext4, vfat, ntfs, etc.). The VFS abstracts these filesystems into a single programming interface
4. The Network Protocol Stack -- High-quality implementation of TCP/IP protocols
5. I/O -- Block and Character Devices
6. And more... (Sound, Virtualization, IPC, etc.)

The Kernel as a Monolithic Architecture

- In the Linux Kernel, all Kernel Components live in the same address space (Also known as the Kernel Segment)
- The Kernel Segment cannot be directly accessed by programs in the user space.
- The entire Kernel needs to be compiled as a single program. (time-consuming)
- Any new code compiled with the Kernel will have the same resource access as the Kernel's major subsystems.

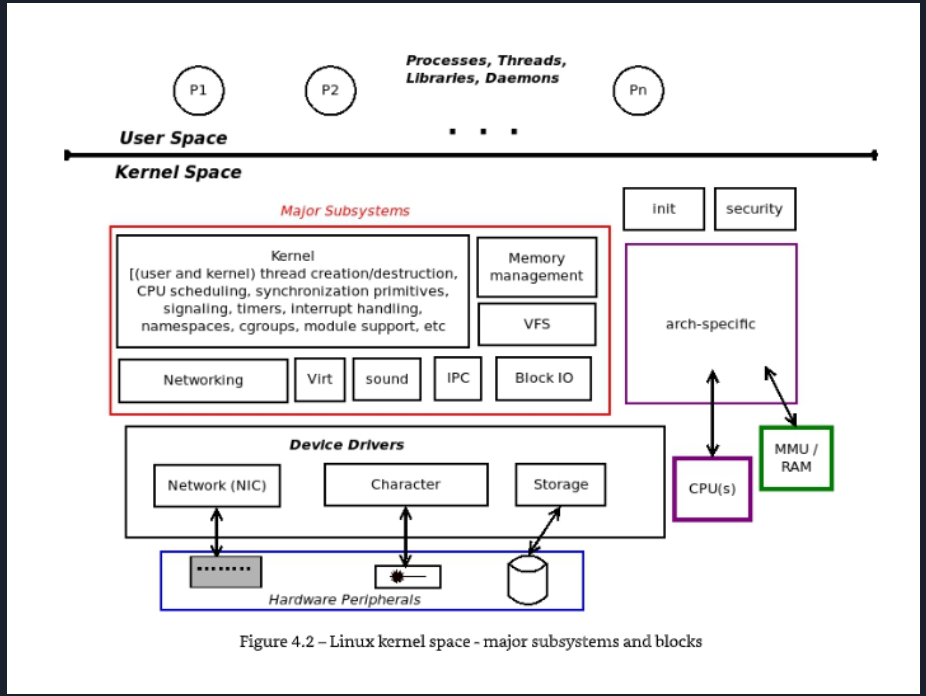


Figure 4.2 – Linux kernel space - major subsystems and blocks

Then how do you add new Kernel functionality (like system drivers) to an existing system?

Loadable Kernel Modules!

Loadable Kernel Modules (LKMs)

- An interface for compiling Kernel code outside of the Kernel's primary source tree.
- Can be loaded into or removed from a system dynamically at runtime.
- Useful for writing hardware-specific code, such as device drivers.
- LKMs exist in the Kernel Space, but do not interact directly with the other major Kernel subsystems.
- Kernel Modules only have access to a subset of the full Kernel API -- Cannot modify some core Kernel functionality (CPU scheduler, timer, or interrupt code).

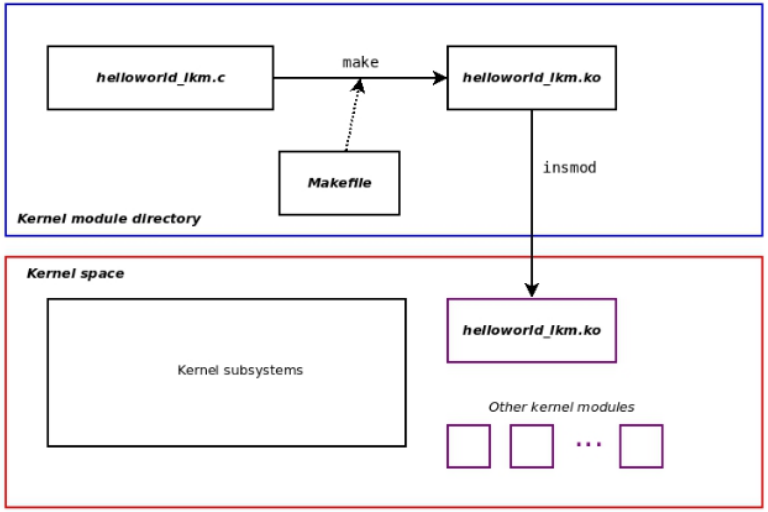


Figure 4.3 – Building and then inserting a kernel module into kernel memory

Worry not: the actual code for both the kernel module C source as well as its Makefile is dealt with in detail in an upcoming section; for now, we want to gain a conceptual understanding only.

How are LKMs loaded into the Kernel?

- LKMs take the form of binary files with the extension *.ko. These files are equivalent to C .o files.
- Kernel modules live in the /lib/modules/[Kernel Version]/kernel/ folder on most Linux systems.
- *insmod* -- “**Ins**ert **Mod**ule”. This is a bash command that takes a Kernel Object File and loads it into the running kernel.
- rmmod -- “**Rem**ove **Mod**ule”. This is a bash command that takes a running Kernel Module and removes it from the running Kernel.
- With these commands, the Kernel can be modified dynamically at runtime!

Steps to Writing a Kernel Module

1. Download and Compile the desired Linux Kernel version
2. Write your Kernel module.
    a. Call Module Information Macros
    b. Define Program Entry Point Function
    c. Define Program Exit Point (Cleanup) Function
    d. Set your functions as the default entry and exit points with the appropriate API calls.
3. Compile your Kernel module into a .ko (kernel object) file
4. Place your file into the kernel modules directory
5. Run “insmod” to insert the kernel module into running memory

Kernel Module “Hello, World!”

```
// ch4/helloworld_lkm/hellowworld_lkm.c
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/module.h>

MODULE_AUTHOR("<insert your name here>");
MODULE_DESCRIPTION("LLKD   book:ch4/helloworld_lkm:   hello,
world, our first LKM");
MODULE_LICENSE("Dual MIT/GPL");
MODULE_VERSION("0.1");

static int __init helloworld_lkm_init(void)
{
    printk(KERN_INFO "Hello, world\n");
    return 0;     /* success */
}

static void __exit helloworld_lkm_exit(void)
{
    printk(KERN_INFO "Goodbye, world\n");
}

module_init(helloworld_lkm_init);
module_exit(helloworld_lkm_exit);
```

Kernel Macros for defining module information.

The Module Entry Point

(Called by insmod)

Kernel Message Buffer Print Function (Like Printf, but logs the information to the Kernel)
Accessed with the dmesg command

The Module Exist Point
(Called by rmmod)

Used for Clean-up

Macro for setting the entry point function

Macro for setting the exist point function

Common Kernel Module Functions

- **kmalloc(size_t size, gfp_t flags)** -- The Kernel equivalent to malloc. Allocates memory in the Kernel Virtual Address Space
- **kfree(const void * objp)** -- Frees memory allocated by kmalloc
- **cdev_open, cdev_release, cdev_read, cdev_write** -- Character device access functions (Operate on the file representing the character device. Used in device drivers). Similar functions exist for Block devices
- **add_timer** -- Start a timer. Takes a timer_list structure as input
- And more! The LKM API is massive!

Questions?