Lecture 17: Introduction to Linux Kernel

Fall 2019
Jason Tang

Slides based upon Linux Device Drivers, 3rd Edition
http://lwn.net/Kernel/LDD3/
Topics

- Linux-specific Filesystems
- Linux Kernel Source Tree Layout
- Writing Kernel Modules
Linux Kernel

- Initially created in 1991 by Linus Torvalds, as a hobby during college
- Supported by numerous companies, often for embedded and specialized systems
- Now used by billions of people, via Android
- Over 25 million lines of code spread across over 62,000+ files
Linux Distributions

• Although there is just one official Linux kernel, various people/groups package the kernel with system libraries, system utilities, and common applications to form entire distributions.

• Thousands of distributions:
  
  • Red Hat: Fedora, Red Hat Enterprise Linux (RHEL), CentOS
  
  • Debian: Debian, Ubuntu, Mint
  
  • Gentoo: Gentoo, Chrome OS
  
  • SUSE: openSuse, SUSE Linux Enterprise
  
  • Google: Android
Filesystem Hierarchy Standard (FHS)

- Most distributors have agreed upon general layout of a running Linux system
  - `/bin, /lib`: essential binaries and libraries
  - `/etc`: host-specific system configuration
  - `/media, /mnt`: mount point for removable media or temporary filesystems
  - `/opt`: add-on application software
  - `/tmp`: temporary files
Pseudo-filesystems

• Linux represents almost everything as a file within the filesystem
  • Not everything actually stored on hard disk

• Memory-backed files: “files” stored entirely in RAM
  • /dev (via devfs), /tmp (via tmpfs), shared memory (via shmfs), other ramfs

• Virtual files: “files” dynamically generated by kernel during file operations
  • /proc (via procfs), /sys (via sysfs), /sys/debug (via debugfs)
/dev

- Traditional location for permanently attached devices (as compared to hotplug devices)

- Accessing a file within invokes matching kernel driver
  - /dev/mem, /dev/null, /dev/zero: calls into kernel’s memory driver
  - /dev/random, /dev/urandom: calls into kernel’s random driver

- Most devices implement file reading and/or writing
  - “Reading” /dev/random returns random numbers from kernel’s PRNG pool
Device Classes

• **Character (char) device**: expressed as a stream of bytes, volatile storage
  • Examples: /dev/console, /dev/random, /dev/ttyS0

• **Block device**: I/O must be transferred in one or more whole blocks, often in increments of 512 bytes, non-volatile storage
  • Examples: /dev/loop0, /dev/sda, /dev/sr0

• **Network device**: network I/O via kernel’s networking system
  • No matching entry within /dev
/proc

- Holds process information and other control information

- Most of these virtual files implement reading, some also implement writing
  - /proc/PID/status: returns status of process PID
  - /proc/self: symbolic link to current process’s PID directory
  - /proc/interrupts: returns all installed interrupt handlers and number of interrupts that have been serviced
  - /proc/slabinfo: returns state of kernel’s slab pools
/sys

- Holds state of kernel subsystems and device drivers

- Like /proc, reading virtual files in /sys within returns current state, while writing changes state

- Modern location to hold driver states
  - Older kernels used to cram everything in /proc
  - **Hotplug devices** represented in /sys
Kernel Subsystems

- Process management
- Memory management
- Filesystems
- Device control
- Networking
Kernel Source

• Linux kernel source code now controlled via git revision control system
  
  • Stable branches are named linux-4.9.y, linux-4.14.y, etc
  
  • Next release of kernel is on master branch
  
  • Linus Torvalds normally controls what gets pushed onto the master branch
  
  • Other major developers maintain stable branches (e.g., Greg Kroah-Hartman)

• **Linux kernel written entirely in C** (not C++, Java, Lua, Python, Ruby, …)
Kernel Source Code Layout

- /Documentation: lots of information about kernel development, coding style, notes about specific hardware

- /arch: architecture-dependent code

- /drivers: device drivers

- /include: header files
  - /include/linux: internal kernel header files
  - /include/UAPI/linux: user space API, which are headers exported to user programs
Kernel Source Code Layout

- `/kernel`: core kernel code (scheduler, thread synchronization)
- `/lib`: common kernel data structures and other code (linked list, trees)
- `/net`: networking code (TCP, UDP, IPv6)
- `/scripts`: kernel build system
Kernel Module (kmod)

• Chunk of code that may be added to kernel at runtime to extend functionality

• All kernel code written in C89, which means:
  
  • Comments must be of form /* … */ (slash-star)
  
  • // (slash-slash) style is not permitted

• Variables must be declared at top of functions, not intermixed with code

• No variable-length arrays

• No floating-point arithmetic
Kernel Module Example

```c
#include <linux/init.h>
#include <linux/module.h>
MODULE_LICENSE("GPL");

static int hello_init(void)
{
    printk(KERN_ALERT "Hello, world\n");
    return 0;
}

static void hello_exit(void)
{
    printk(KERN_ALERT "Goodbye, cruel world\n");
}

module_init(hello_init);
module_exit(hello_exit);
```
Writing Kernel Modules

• Similar to C user software, in that there is a single entry point and exit point

  • Module is responsible for cleaning up after itself during exit

• No standard library; only header files in kernel source tree’s /include may be used

• Typically designed as event-driven application:

  • During initialization, module registers callbacks into core kernel code

  • When event occurs, core kernel invokes callback

  • goto is used for error handling
Concurrencym

- Linux kernel internally is multithreaded (via kthreads)

- Modules can be preempted, or can also be invoked concurrently
  - Example: Multiple user space processes can read from /dev/random simultaneously

- Callbacks must use synchronization to avoid race conditions
  - mutex, semaphore, and/or condition variable
  - futex, rcu_lock, spinlock
Compiling Kernel Modules

• Module could be compiled as part of Linux kernel (an in-tree module) or in separate directory (an out-of-tree module)

• Either way requires kernel build system (Kbuild) support

• Typically, Kbuild will compile a file named foo.c into the kernel module foo.ko

• Load a module via insmod: insmod foo.ko

• Remove a module via rmmod: rmmod foo
Module Initialization

- Entry point into module, called by core kernel when module is inserted

- Nearly always declared as `static` and has `__init` token

  - Hint to kernel that function is only used at initialization, and can be purged from memory afterwards

- Use `module_init()` macro to declare which function is for initialization

```c
static int __init hello_init(void) {
    printk(KERN_ALERT "Hello, world\n");
    return 0;
}
module_init(hello_init);
```
Module Initialization

```c
static int __init hello_init(void)
{
    printk(KERN_ALERT "Hello, world\n");
    return 0;
}

module_init(hello_init);
```

- Function returns 0 on successful initialization, or negative on error

- Return the negative of an `errno` value:
  - `ENOMEM`: out of memory
  - `EPERM`: operation not permitted
Displaying Messages

```c
static int __init hello_init(void)
{
    printk(KERN_ALERT "Hello, world\n");
    return 0;
}

module_init(hello_init);
```

• Classically, use `printk()` to generate messages

  • Superseded in newer kernels by `pr_*()` and `dev_*()` functions

• Messages sent to kernel log

• Use `dmesg` command to view contents of kernel log
Module Shutdown

- Invoked by core kernel when module is unloaded
- Responsible for releasing memory, unlocking locks, etc.
  - You are responsible for cleaning up after yourself
- Normally declared as both static and with __exit token
- Use module_exit() macro to declare which function is for cleanup

```c
static void hello_exit(void) {
    printk(KERN_ALERT "Goodbye, cruel world\n");
}
module_exit(hello_exit);
```