CSCI 3753
Operating Systems

Kernel Programming

Lecture Notes By

Shivakant Mishra

Computer Science, CU-Boulder

Last Update: 09/12/2017

Kernel Programming Environment

- Kernel has no libc, none of the standard headers and libraries you are used to.
  - i.e. No system calls, no standard buffered output (stdio.h)
- No memory protection!
  - You can stomp on any other part of memory, no one will stop you.

Kernel Programming Environment

- One big single namespace
  - You have access to all “exported” symbols from your kernel module.
  - Exported functions are called “kernel services”
- Always multi-threaded
  - All modules must be thread-safe, like it or not
  - What about single-processor systems?
    - No concurrency, but Linux is a preemptable kernel! Still must be thread-safe.

Linux Internals

- We will study
  - proc file system
  - printk( ) and dmesg
  - Loadable kernel module
  - Device drivers as LKM – recitations and programming assignment two

- System call interception
- Kprobes
- Linux security module

Later in the semester if time permits

/proc directory

- Also called proc file system: mechanism for kernel and kernel modules to send info to processes in user space
- Contains a hierarchy of special files which represent the current state of the kernel
- Allows applications and users to peer into the kernel's view of the system
- A wealth of information detailing the system hardware and any processes currently running
- Some of the files can be manipulated by users and applications to communicate configuration changes to the kernel.

/proc: Virtual file system

- /proc/ directory contains a third type of file called a *virtual file* (besides binary and text files)
- Typically listed as zero byte size, and yet contain a large amount of information
- Most of the time and date settings on virtual files reflect the current time and date, indicative of the fact they are constantly updated
- For organizational purposes, files containing information on a similar topic are grouped into virtual directories and sub-directories
  - e.g. /proc/ide/ contains information for all physical IDE devices

procfs

- On /proc is usually mounted what is known as procfs – special file systems
- Open/read/write … requests get passed to the procfs file system driver code, which knows about all these files and directories *and returns particular pieces of information from the kernel data structures*
- The "storage layer" in this case is the kernel data structures, and procfs provides a clean, convenient interface to accessing them

Viewing Virtual Files

- By using the cat, more, or less commands on files within the /proc/ directory
- Some of the information is easily understandable, some is not human-readable, some can be read only by root users
- Utilities exist to pull data from virtual files and display it in a useful way
  - lspci, apm, free, and top
  - *Read man pages*

Changing virtual files

- Most virtual files within the /proc/ directory are read-only
- Some can be used to adjust settings in the kernel
  - E.g. files in the /proc/sys/ subdirectory
- To change the value of a virtual file, use the echo command and a greater than symbol (>) to redirect the new value to the file
  - E.g. *echo www.example.com > /proc/sys/kernel/hostname*
- Some files act as binary or Boolean switches
  - E.g. cat /proc/sys/net/ipv4/ip_forward
  - 0: not forwarding IP packets
- Another way to alter settings in the /proc/sys/ subdirectory is /sbin/sysctl

Top-level files in proc file system

- /proc/cmdline: shows the parameters passed to the kernel at the time it is started
- /proc/cpuinfo: identifies the type of processor used by your system
- /proc/crypto: lists all installed cryptographic ciphers used by the Linux kernel
- /proc/devices: displays the various character and block devices currently configured
- /proc/filesystems: displays a list of the file system types currently supported by the kernel
- /proc/iomem: shows you the current map of the system's memory for each physical device
- Several others ...

printk( )

- In kernel mode, you can’t use standard C library, so printf( ) is not available
- printk( ) is a function that prints messages to kernel log, and is used in C exclusively for the Linux kernel
  - int printk(const char *fmt, ...);
- parsing of the format string and arguments behave like printf( )
- printk() can be called from anywhere in the Kernel at any time. It can be called from interrupt or process context

printk( ): Logging levels

- printk has an optional prefix string, Loglevel, that specifies the type of message being sent to the kernel message log

KERN_EMERG	Emergency condition, system is probably dead
KERN_ALERT	Some problem has occurred, immediate attention is needed
KERN_CRIT	A critical condition
KERN_ERR	An error has occurred
KERN_WARNING	A warning
KERN_NOTICE	Normal message to take note of
KERN_INFO	Some information
KERN_DEBUG	Debug information related to the program

dmesg

- *dmesg* (*display message* or *driver message*) is a command that prints the message buffer of the kernel.
- When initially booted, the kernel produces messages reporting both the presence of modules and the values of any parameters adopted
- The booting process typically happens quite fast
- The dmesg command allows the review of such messages in a controlled manner after the system has started

Loadable Kernel Modules

- A loadable kernel module (LKM) is an object file that contains code to extend a running kernel
- Windows (kernel-mode driver), Linux (LKM), OS X (Kernel extension: kext), VmWorks (downloadable kernel module: DKM), …
- Without loadable kernel modules, an OS would have to include all possible anticipated functionality already compiled directly into the base kernel – monolithic kernel
- LKMs can be loaded and unloaded from kernel on demand at runtime

LKMs

- Offer an easy way to extend the functionality of the kernel without having to rebuild or recompile the kernel again
- Simple and efficient way to create programs that reside in the kernel and run in privileged mode
- Most of the drivers are written as LKMs
- See */lib/modules* for the all the LKMs
- *lsmod*: lists all kernel modules that are already loaded
  - Reads */proc/modules* file

How to write a kernel module

- Kernel Modules are written in the C programming language.
- You must have a Linux kernel source tree to build your module.
- You must be running the same kernel you built your module with to run it.
- Linux kernel object: *.ko* extension

Kernel Module: Basics

- A kernel module file has several typical components:
  - MODULE_AUTHOR(“your name”)
  - MODULE_LICENSE(“GPL”)
    - The license must be an open source license (GPL, BSD, etc.) or you will “taint” your kernel.
    - Tainted kernel loses many abilities to run other open source modules and capabilities.

Kernel Module: Key Operations

- int init_module(void)
  - Called when the kernel loads your module.
  - Initialize all your stuff here.
  - Return 0 if all went well, negative if something blew up.
- Typically, init_module( ) either registers a handler for something with the kernel, or replaces one of the kernel functions with its own code (usually code to do something and then call the original function)

- void cleanup_module(void)
  - Called when the kernel unloads your module.
  - Free all your resources here.

Hello World Example

```
#include <linux/kernel.h>
#include <linux/module.h>
MODULE_AUTHOR("Shiv Mishra");
MODULE_LICENSE("GPL");


 int init_module(void)
 {
 printk(KERN_ALERT "Hello world: I am Shiv Mishra
                                   speaking from the Kernel");
 return 0;
 }
```

Hello World Example

```
void cleanup_module(void)
{
printk(KERN_ALERT “Goodbye from Shiv Mishra,
                                        I am exiting the Kernel”);
}
```

Building Your Kernel Module

- Accompany your module with a 1-line GNU Makefile:
  – obj-m += hello.o
  – Assumes file name is “hello.c”
- Run the magic make command:
  – make -C <kernel-src> M=`pwd` modules
  – Produces: hello.ko
- Assumes current directory is the module source.

obj-$(CONFIG_FOO) += foo.o

- Good definitions are the main part (heart) of the kbuild Makefile.
- The most simple kbuild makefile contains one line:

```
obj-$(CONFIG_FOO) += foo.o
```

- This tell kbuild that there is one object in that directory named foo.o. foo.o will be built from foo.c or foo.S.
- $(CONFIG_FOO) evaluates to either y (for built-in) or m (for module). If CONFIG_FOO is neither y nor m, then the file will not be compiled nor linked.

Loading Your Kernel Module: *insmod*

- *Use insmod* to manually load your kernel module

  sudo insmod helloworld.ko

- *insmod* makes an *init_module* system call to load the LKM into kernel memory
- *init_module* system call invokes the LKM's initialization routine (also called *init_module*) right after it loads the LKM
- The LKM author sets up the initialization routine to call a kernel function that registers the subroutines that the LKM contains

Where is our Hello World message

- dmesg

Unloading Your Kernel Module

- Use *rmmod* command

  rmmod hello.ko

- Should print the Goodbye message

Kernel Module Dependencies: *modprobe*

- insmod/rmmod can be cumbersome...
  - You must manually enforce inter-module dependencies.
- *modprobe* automatically manages dependent modules
  - Copy hello.ko into /lib/modules/<version>
  - Run depmod
  - modprobe hello / modprobe -r hello
- Dependent modules are automatically loaded/unloaded.

- *depmod* creates a Makefile-like dependency file, based on the symbols it finds in the set of modules mentioned on the command line or from the directories specified in the configuration file
- This dependency file is later used by *modprobe* to automatically load the correct module or stack of modules

modinfo command

- .ko files contain an additional .modinfo section where additional information about the module is kept
  - Filename, license, dependencies, …
- modinfo command retrieves that information

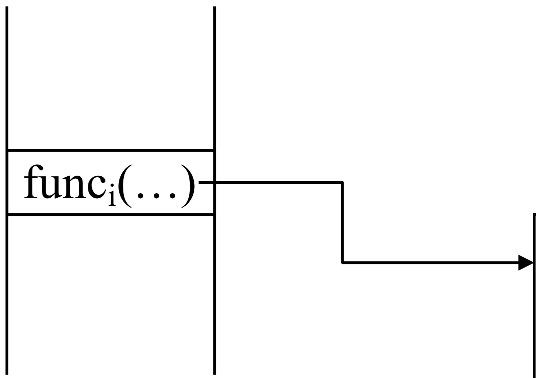
  modinfo hello.ko

Reconfigurable Device Drivers

- Allows system administrators to add a device driver to the OS without recompiling the OS
- The new driver is first stored as a *.ko* file
  - Contains an initialization routine
- The initialization routine calls a kernel function to register the device
  - e.g. *register_chrdev*, *register_blkdev*

Device Independent Function Call

Trap Table




```
dev_func_i(devID, …) {
// Processing common to all devices
   …
   switch(devID) {
   case dev0:  dev0_func_i(…);
               break;
   case dev1:  dev1_func_i(…);
               break;
   …
   case devM:  devM_func_i(…);
               break;
   };
// Processing common to all devices
   …
}
```

- An entry table stores the actual function pointers for each device specific function call

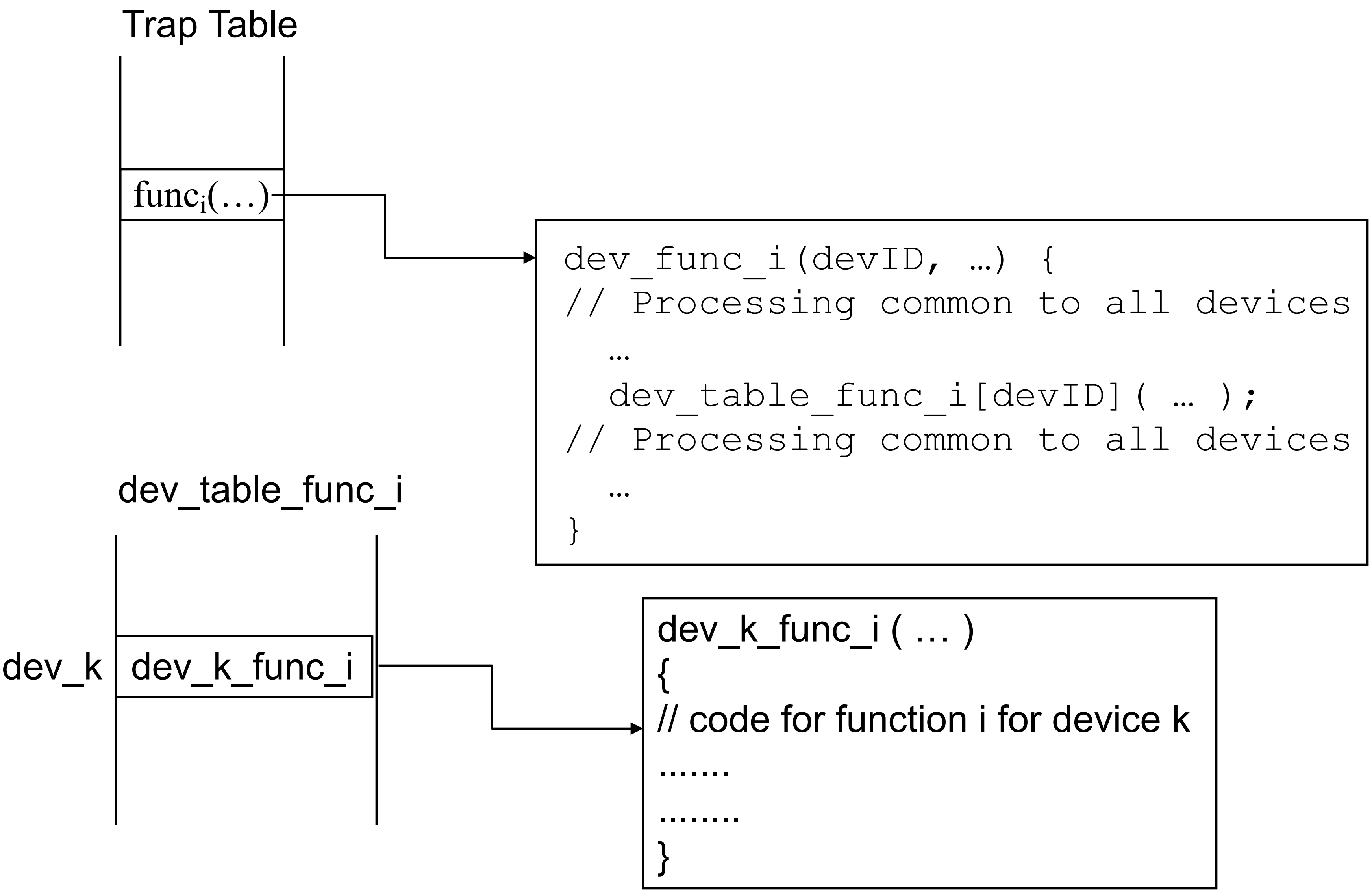
  dev_func_i[N]

- Replace *switch* statement with

  dev_func_i[j] (…);

- Device registration: Fill appropriate function pointers in the entry table

Device Independent Function Call



- Device Driver as LKM

  … details will be covered in recitations …

  programming assignment two