Linux Kernel Module Programming

Amir H. Payberah
amir@sics.se

Amirkabir University of Technology
(Tehran Polytechnic)

Introduction

What Is A Kernel Module?

- Pieces of code that can be loaded and unloaded into the kernel upon demand.

What Is A Kernel Module?

- Pieces of code that can be loaded and unloaded into the kernel upon demand.

- They extend the functionality of the kernel without the need to reboot the system.

What Is A Kernel Module?

- Pieces of code that can be loaded and unloaded into the kernel upon demand.

- They extend the functionality of the kernel without the need to reboot the system.

- One type of module is the device driver.

Already Loaded Modules

- You can see already loaded modules into kernel: `lsmod`
- It gets its information by reading the file `/proc/modules`.

How Do Modules Get Into The Kernel? (1/2)

- When the kernel needs a feature that is not resident in the kernel:
- The kernel module daemon `kmod` execs `modprobe` to load the module in.

How Do Modules Get Into The Kernel? (1/2)

- When the kernel needs a feature that is not resident in the kernel:
- The kernel module daemon `kmod` execs `modprobe` to load the module in.
- `modprobe` looks file `/lib/modules/<version>/modules.dep`.
  - Contains module dependencies
  - Created by `depmod -a`

How Do Modules Get Into The Kernel? (2/2)

- `modprobe` uses `insmod` to load any prerequisite modules and the requested module into the kernel.

How Do Modules Get Into The Kernel? (2/2)

- modprobe uses insmod to load any prerequisite modules and the requested module into the kernel.
- insmod is dumb about the location of modules, whereas modprobe is aware of the default location of modules.

How Do Modules Get Into The Kernel? (2/2)

- `modprobe` uses `insmod` to load any prerequisite modules and the requested module into the kernel.
- `insmod` is dumb about the location of modules, whereas `modprobe` is aware of the default location of modules.
- `modprobe` also knows how to figure out the dependencies and load the modules in the right order.

insmod vs. modprobe

- `insmod` requires you to pass it the full pathname and to insert the modules in the right order.

insmod vs. modprobe

- `insmod` requires you to pass it the full pathname and to insert the modules in the right order.
- `modprobe` just takes the name, without any extension, and figures out all it needs to know.

insmod vs. modprobe

- `insmod` requires you to pass it the full pathname and to insert the modules in the right order.
- `modprobe` just takes the name, without any extension, and figures out all it needs to know.
- For example:

```
insmod /lib/modules/2.6.11/kernel/fs/fat/fat.ko
insmod /lib/modules/2.6.11/kernel/fs/msdos/msdos.ko
```

```
modprobe msdos
```

Hello World

Hello World

```c
#include <linux/module.h> /* Needed by all modules */
#include <linux/kernel.h> /* Needed for KERN_INFO */

int init_module(void) {
  printk(KERN_INFO "Hello world!\n");
  // A non 0 return means init_module failed; module can't be loaded.
  return 0;
}

void cleanup_module(void) {
  printk(KERN_INFO "Goodbye world!.\n");
}
```

Hello World Structure (1/2)

- Two main functions of kernel modules: initialization and cleanup.

Hello World Structure (1/2)

- Two main functions of kernel modules: initialization and cleanup.

- `init_module()`: it is called when the module is insmoded into the kernel.
  - It registers a handler for something with the kernel.
  - or it replaces one of the kernel functions with its own code.

Hello World Structure (1/2)

- Two main functions of kernel modules: initialization and cleanup.
- `init_module()`: it is called when the module is insmoded into the kernel.
  - It registers a handler for something with the kernel.
  - or it replaces one of the kernel functions with its own code.
- `cleanup_module()`: it is called just before it is rmmoded.
  - It undos whatever `init_module()` did, so the module can be unloaded safely

Hello World Structure (2/2)

- Every kernel module needs `linux/module.h`.
- It needs `linux/kernel.h` only for the macro expansion for the `printk()` log level, e.g., `KERN_INFO`.

Introducing `printk()`

- Logging mechanism for the kernel.

Introducing printk()

- Logging mechanism for the kernel.
- Each `printk()` statement comes with a priority.
  - Eight priority levels

Introducing printk()

- Logging mechanism for the kernel.
- Each `printk()` statement comes with a priority.
  - Eight priority levels
- Prints out the message in the kernel ring buffer.
  - Use `dmesg` to print the kernel ring buffer.

Introducing `printk()`

- Logging mechanism for the kernel.
- Each `printk()` statement comes with a priority.
  - Eight priority levels
- Prints out the message in the kernel ring buffer.
  - Use `dmesg` to print the kernel ring buffer.
- The message is appended to `/var/log/messages`, if `syslogd` is running.

Compiling Kernel Modules

- A simple Makefile for compiling a module named `hello.c`.

```
obj-m += hello.o

all:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
clean:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

Compiling Kernel Modules

- A simple Makefile for compiling a module named `hello.c`.

```
obj-m += hello.o

all:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
clean:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

- To compile and insert the module in the kernel:

```
# to compile the module
> make

# to insert the module in the kernel
> sudo insmod hello.ko

# to see the module message
> dmesg

# to remove the module from the kernel
> sudo insmod hello.ko
```

Hello World Version 2

Hello World V. 2

- Renaming the init and cleanup functions with `module_init()` and and `module_exit()` macros.

```
#include <linux/module.h> /* Needed by all modules */
#include <linux/kernel.h> /* Needed for KERN_INFO */
#include <linux/init.h>   /* Needed for the macros */

static int __init hello_init(void) {
  printk(KERN_INFO "Hello, world!\n");

  return 0;
}

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

module_init(hello_init);
module_exit(hello_exit);
```

Hello World Version 3

Passing Command Line Arguments to a Module

```
module_param(name, type, perm)
/*
* The first param is the parameter's name
* The second param is it's data type
* The final argument is the permissions bits
*/

module_param_array(name, type, num, perm);
/*
* The first param is the parameter's
* The second param is the data type of the elements of the array
* The third argument is a pointer to the variable that will store the number
* of elements of the array initialized by the user at module loading time
* The fourth argument is the permission bits
*/
```

Hello World V. 3 (1/2)

```c
#include <linux/module.h>
#include <linux/moduleparam.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/stat.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Peter Jay Salzman");

static int myint = 420;
static int myintArray[2] = { -1, -1 };
static int arr_argc = 0;

module_param(myint, int, S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
MODULE_PARM_DESC(myint, "An integer");

module_param_array(myintArray, int, &arr_argc, 0000);
MODULE_PARM_DESC(myintArray, "An array of integers");
```

Hello World V. 3 (2/2)

```
static int __init hello_init(void) {
  int i;

  printk(KERN_INFO "Hello, world!\n");
  printk(KERN_INFO "myint is an integer: %d\n", myint);

  for (i = 0; i < (sizeof myintArray / sizeof (int)); i++)
    printk(KERN_INFO "myintArray[%d] = %d\n", i, myintArray[i]);

  printk(KERN_INFO "got %d arguments for myintArray.\n", arr_argc);

  return 0;
}

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

module_init(hello_init);
module_exit(hello_exit);
```

Compiling Kernel Modules

```
# to insert the module in the kernel
> sudo insmod hello.ko myint=10 myintArray=10,20
```

Device Drivers

Device Drivers (1/2)

- One class of module is the device driver.
- Each piece of hardware is represented by a file located in `/dev`.
- A device file provides the means to communicate with the hardware.

Device Drivers (2/2)

- For example, the `es1370.o` sound card device driver might connect the `/dev/sound` device file to the Ensoniq IS1370 sound card.

Device Drivers (2/2)

- For example, the `es1370.o` sound card device driver might connect the `/dev/sound` device file to the Ensoniq IS1370 sound card.
- A userspace program like mp3blaster can use `/dev/sound` without ever knowing what kind of sound card is installed.

Major and Minor Numbers

- The major device number identifies a device driver that should be called.
- The minor device number is passed to the device driver, and it is entirely up to the driver how the minor number is being interpreted.

```
> ls -l /dev/sda*

brw-rw---- 1 root disk 8, 0 Dec  4 19:50 /dev/sda
brw-rw---- 1 root disk 8, 1 Dec  4 19:50 /dev/sda1
brw-rw---- 1 root disk 8, 2 Dec  4 19:50 /dev/sda2
brw-rw---- 1 root disk 8, 3 Dec  4 19:50 /dev/sda3
brw-rw---- 1 root disk 8, 4 Dec  4 19:50 /dev/sda4
brw-rw---- 1 root disk 8, 5 Dec  4 19:50 /dev/sda5
brw-rw---- 1 root disk 8, 6 Dec  4 19:50 /dev/sda6
brw-rw---- 1 root disk 8, 7 Dec  4 19:50 /dev/sda70
```

Creating A Device File

- All of device files are created by the `mknod` command.
- For example, to create a new char device named coffee with major/minor number 12 and 2.

```
> mknod /dev/coffee c 12 2
```

Character Device Drivers

The file_operations Structure (1/2)

- It holds pointers to functions defined by the driver that perform various operations on the device.
- Defined in linux/fs.h.

```
struct file_operations {
  struct module *owner;
  int (*open)(struct inode *, struct file *);
  ssize_t (*read)(struct file *, char __user *, size_t, loff_t *);
  ssize_t (*write)(struct file *, const char __user *, size_t, loff_t *);
  int (*ioctl)(struct inode *, struct file *, unsigned int, unsigned long);
  int (*mmap)(struct file *, struct vm_area_struct *);
  int (*flush)(struct file *);
  int (*release)(struct inode *, struct file *);
  int (*lock)(struct file *, int, struct file_lock *);
  unsigned int (*poll)(struct file *, struct poll_table_struct *);
  ...
};
```

The file_operations Structure (2/2)

- The common syntax to use file_operations structure.

```
struct file_operations fops = {
  .read = device_read,
  .write = device_write,
  .open = device_open,
  .release = device_release
};
```

The `file` Structure

- Each device is represented in the kernel by a `file` structure, which is defined in `linux/fs.h`.
- The file is a kernel level structure and never appears in a user space program.
  - It is not the same thing as a FILE.

```
struct file {
  mode_t f_mode;
  loff_t f_pos;
  unsigned int f_flags;
  struct file_operations *f_op;
  void *private_data;
  ...
};
```

Registering A Device

- Adding a driver to the system.
- Using the `register_chrdev` function, defined by `linux/fs.h`.

```
int register_chrdev(unsigned int major, const char *name,
                    struct file_operations *fops);
```

Unregistering A Device

- Keeps track of the number of processes using the module.
  - 3rd field of `/proc/modules`
  - If this number is not zero, `rmmod` will fail.

```
#include<linux/module.h>

// Increment the use count
try_module_get(THIS_MODULE)

// Decrement the use count
module_put(THIS_MODULE)
```

put_user and get_user

- put_user() and get_user() are used to get and put single values (such as an int, char, or long) from and to userspace.

Character Driver Example (1/7)

```c
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/fs.h>
#include <asm/uaccess.h> /* for put_user */
/*
* Prototypes - this would normally go in a .h file
*/
int init_module(void);
void cleanup_module(void);
static int device_open(struct inode *, struct file *);
static int device_release(struct inode *, struct file *);
static ssize_t device_read(struct file *, char *, size_t, loff_t *);
static ssize_t device_write(struct file *, const char *, size_t, loff_t *);

#define SUCCESS 0
#define DEVICE_NAME "chardev" /* Dev name as it appears in /proc/devices */
#define BUF_LEN 80 /* Max length of the message from the device */
```

Character Driver Example (2/7)

```c
/*
* Global variables are declared as static, so are global within the file.
*/
static int Major;           /* Major number assigned to our device driver */
static int Device_Open = 0; /* Is device open?
                             * Used to prevent multiple access to device */
static char msg[BUF_LEN];   /* The msg the device will give when asked */
static char *msg_ptr;

static struct file_operations fops = {
  .read = device_read,
  .write = device_write,
  .open = device_open,
  .release = device_release
};
```

Character Driver Example (3/7)

```
int init_module(void) {
  Major = register_chrdev(0, DEVICE_NAME, &fops);
  if (Major < 0) {
    printk(KERN_ALERT "Registering char device failed with %d\n", Major);
    return Major;
  }

  printk(KERN_INFO "I was assigned major number %d. To talk to\n", Major);
  printk(KERN_INFO "'mknod /dev/%s c %d 0'.\n", DEVICE_NAME, Major);

  return SUCCESS;
}

void cleanup_module(void) {
  unregister_chrdev(Major, DEVICE_NAME);
}
```

Character Driver Example (4/7)

```c
/*
* Called when a process tries to open the device file, like
* "cat /dev/mycharfile"
*/
static int device_open(struct inode *inode, struct file *file) {
  static int counter = 0;
  if (Device_Open)
    return -EBUSY;

  Device_Open++;
  sprintf(msg, "I already told you %d times Hello world!\n", counter++);
  msg_Ptr = msg;

  try_module_get(THIS_MODULE);

  return SUCCESS;
}
```

Character Driver Example (5/7)

```
/*
* Called when a process closes the device file.
*/
static int device_release(struct inode *inode, struct file *file) {
  Device_Open--; /* We're now ready for our next caller */

  /*
  * Decrement the usage count, or else once you opened the file, you'll
  * never get get rid of the module.
  */
  module_put(THIS_MODULE);

  return 0;
}
```

Character Driver Example (6/7)

```
/*
* Called when a process, which already opened the dev file, attempts to
* read from it.
*/
static ssize_t device_read(struct file *filp, /* see include/linux/fs.h */
  char *buffer, /* buffer to fill with data */
  size_t length, /* length of the buffer */
  loff_t * offset) {
  int bytes_read = 0;
  if (*msg_ptr == 0)
    return 0;

  /*
   * The buffer is in the user data segment. We have to use put_user which
   * copies data from the kernel data segment to the user data segment.
   */
  while (length && *msg_ptr) {
    put_user(*(msg_ptr++), buffer++);
    length--;
    bytes_read++;
  }

  return bytes_read;
}
```

Character Driver Example (7/7)

```
/*
* Called when a process writes to dev file: echo "hi" > /dev/hello
*/
static ssize_t
device_write(struct file *filp, const char *buff, size_t len, loff_t *off) {
  printk(KERN_ALERT "Sorry, this operation isn't supported.\n");
  return -EINVAL;
}
```

Talking To Device Files

- `ioctl` (input-output contro) is a device-specific system call.
- There are only a few system calls in Linux, which are not enough to express all the unique functions devices may have.
- So a driver can define an `ioctl` that allows a userspace application to send it orders.

ioctl Example (1/3)

```
struct file_operations Fops = {
  .read = device_read,
  .write = device_write,
  .ioctl = device_ioctl,
  .open = device_open,
  .release = device_release, /* a.k.a. close */
};
```

ioctl Example (2/3)

```
#define MAJOR_NUM 100
#define IOCTL_SET_MSG _IOR(MAJOR_NUM, 0, char *)
#define IOCTL_GET_MSG _IOR(MAJOR_NUM, 1, char *)

int device_ioctl(struct inode *inode, struct file *file,
  unsigned int ioctl_num, /* number and param for ioctl */
  unsigned long ioctl_param) {
  char *temp;

  switch (ioctl_num) {
    case IOCTL_SET_MSG:
      temp = (char *)ioctl_param;
      get_user(ch, temp);
      ...
      break;
    case IOCTL_GET_MSG:
      ...
      break;
  }

  return SUCCESS;
}
```

ioctl Example (3/3)

```
main() {
  int fd;
  int ret_val;
  char *msg = "Message passed by ioctl\n";
  char message[100];

  fd = open("/dev/chardev", 0);

  ret_val = ioctl(fd, IOCTL_GET_MSG, message);
  printf("get_msg message:%s\n", message);

  ret_val = ioctl(fd, IOCTL_SET_MSG, msg);

  close(file_desc);
}
```

Summary

Summary

- Kernel modules

Summary

- Kernel modules
- `insmod` and `modprobe`

Summary

- Kernel modules
- `insmod` and `modprobe`
- `init_module` and `cleanup_module`

Summary

- Kernel modules
- insmod and modprobe
- init_module and cleanup_module
- printk

Summary

- Kernel modules
- `insmod` and `modprobe`
- `init_module` and `cleanup_module`
- `printk`
- `module_param` and `module_param_array`

Summary

- Kernel modules
- insmod and modprobe
- init_module and cleanup_module
- printk
- module_param and module_param_array
- Device drivers: major and minor numbers

Summary

- Kernel modules
- insmod and modprobe
- init_module and cleanup_module
- printk
- module_param and module_param_array
- Device drivers: major and minor numbers
- file_operations

Questions?