An Introduction to Linux Kernel Programming with Modules

john.justice@trilogy.com

3/28/01

Contents:

- **What makes kernel coding different?**

- **Types of kernel drivers**
  - char, block, scsi, net
  - Loaded as modules or static in the kernel

- **Challenges**
  - Portability
  - IPC
  - Hardware Management
  - Interface Stability / Documentation

Basic Kernel Module: hello.c


```
#define MODULE                          /* required for all modules */
#include <linux/module.h>               /* required for all modules */

int init_module(void)                   /* called at insmod */
{
    printk("Hello World\n");
    return 0;                           /* success */
}

void cleanup_module(void)               /* called at rmmod */
{
    printk("Goodbye\n");
}
```

Basic Kernel Module: hello.c

- **Why are we using printk()?**
  - Kernel space code can't use any user space libraries.
  - printk() allows you to specify priority: KERN_DEBUG, KERN_INFO, KERN_WARNING, KERN_EMERG, etc.
  - Usage: printk(priority message, args); /* Like printf() with a priority */

- **How do I use this module?**
  - "# insmod module_name" will load your module
  - "# rmmod module_name" will remove it
  - depmod, modprobe, kmod all let you automate the process

- **I don't see any the "Hello World" and "Goodbye" messages**
  - The default printk() priority isn't high enough to show up on most consoles. Either up the priority, or view the messages with "$ dmesg".

Getting more advanced: hello_bob.c


```
static int nums[MAX_NUMBERS] = {0, };       /* default values*/
static char *name[1] = {“Bob”};
static char *colors[3] = {“red”, “green”, “orange”};

MODULE_PARM(nums, “1-4”__MODULE_STRING( MAX_NUMBERS) “i”);
                                       /* int, min=1, max=4 */
MODULE_PARM(name, “s”);                /* string, min = 1, max =1 */
MODULE_PARM(colors, “2-3s”);           /* string, min=2, max=3 */

int init_module(void)
{
    printk(KERN_INFO “Hello %s!\n”, name[0]);
    printk(“Your favorite colors are %s, %s, and %s.\n”,
                colors[0], colors[1], colors[2]);
    printk(“Your lucky numbers are %i, %i, %I,  and %i.\n”,
                nums[0], nums[1], nums[2], nums[3]);
    return 0;
}
```

- **What about Arguments?**
  - Arguments handling has changed a bit in the last few years.
  - MODULE_PARM() lets us register our arguments of type int, long, char, or string. Repeated arguments are handled with a "min-max" syntax.

    "# insmod hello_bob name=Willie colors=brown,black nums=1,2,3"

    If the argument count isn't legal, insmod complains.
  - Note: Setting min to 1 is ignored. See <linux/modules.h> for more details.

- **So now we can start a module with a given set of arguments, but can it do anything useful?**
  - Yes, but we'll need to perform some additional setup tasks first.
    - Create a file_operations structure that declares what we can do
    - Fill in that structure with our operations
    - Handle device registration and unregistration

A Simple char Device: status.c


```
struct file_operations status_fops = {
    NULL,               /* seek */
    read_status,
    write_status,
    NULL,               /* readdir */
    NULL,               /* poll */
    NULL,               /* ioctl */
    NULL,               /* mmap */
    open_status,
    NULL,               /* flush */
    close_status,
    NULL,               /* fsync */
    NULL,               /* fasync */
    NULL,               /* check_media_change */
    NULL,               /* revalidate */
    NULL,               /* lock */
};
```

A Simple char Device: status.c


```
static int major_number = 0;    /* pass a major of 0 to register_chrdev
                                  * for dynamic allocation
                                  */
int init_module(void)
{
    major_number = register_chrdev(0, "status", &status_fops);
    if (major_number < 0) {
        printk(KERN_WARNING "Dynamic allocation of major failed");
        return major_number;    /* return error code */
    }
    printk(KERN_INFO "Assigned major number %i.\n", major_number);

    return 0;
}
```

- **To use this device, we'll have to make an entry in /dev for its major number: "mknod /dev/status0 c <major_num> 0"**

```
int cleanup_module(void)
{
    printk(KERN_INFO "Unregistering major number %i.\n", major_number);
    unregister_chrdev(major_number, "status"); /* give back our number */

    return 0;
}

static ssize_t write_status(struct file *file, const char *buffer, size_t count,
                loff_t *ppos)
{
    return -EINVAL;  /* we've decided not to support writes for now */
}
```

A Simple char Device: status.c


```
static int open_status(struct inode *inode, struct file *file)
{
   MOD_INC_USE_COUNT; /* ensures that currently used  modules aren't
                       * unloaded
                       */
   return 0;
}

static int close_status(struct inode *inode, struct file *file)
{
   MOD_DEC_USE_COUNT; /* rmmod won't run unless USE_COUNT is 0 */

   return 0;
}
```

A Simple char Device: status.c

TRILOGY

```
static char *message = "All your base are belong to us.";

static ssize_t read_status(struct file *file, char *buffer, size_t count, loff_t
        *ppos)
{
    int char_count = 0;
    int count_to_copy = 0;

    while (message[char_count] != '\0') {
        char_count ++;
    }

    count_to_copy = (char_count > count) ? count : char_count;
    copy_to_user(buffer, message, count_to_copy); /* write to the user-
                                                   * space buffer */
    return char_count;
}
```

- **So now we have a working character device that tells us the status of our base. To make it more useful, we're going to need some dynamically allocated memory.**

- **There are a few functions we care about:**
  - kmalloc(), kfree(), vmalloc(), vfree(), get_free_page()

  - kmalloc(): Allocates contiguous physical ranges, suitable for DMA. At high priorities, this is non-blocking.
  - kfree(): Frees those same ranges
  - vmalloc(): Allocates contiguous logical ranges, good for large software buffers. Won't work for DMA.
  - vfree(): Not surprisingly, vfree releases ranges grabbed with vmalloc
  - get_free_page(): When you want to do all the work yourself, use this. Grabs actual physical pages for your sole use.

Sooner or later you'll want your device driver to do something with devices.

- **Device probing**
  - Finding PCI devices under Linux is quick and painless. The pcibios_xxx class of calls make the job a breeze.
  - ISA probing is painful and evil. Avoid it if you can.

- **Linux supports both port I/O and memory-mapped I/O**
  - inb(), outb(), inw(), outw(), inl(), outl() perform single value transfers to and from I/O ports
  - insb(), outsb(), insw(), outsw(), insl(), outsl() perform string transfers to and from I/O ports
  - read[bwl], write[bwl] perform single value transfers to and from I/O memory.

Porting your code between architectures can be an intensely painful experience. Here are some areas to watch.

- **Data Types**
  - Use size specific data types when you intend them. Don't assume x86 hardware. u8, u16, u32, u64 etc. are our friends. Use them.
- **Behavioral Assumptions**
  - While you can directly dereference a pointer to hardware in the x86 world, the same assumption doesn't hold true on an Alpha. Make sure you realize your assumptions, before they crash your machine.
- **Magic Numbers**
  - Make sure your code uses PAGE_SIZE and not 4KB.
- **Trust GNU**
  - Compile with -Wall and remove all the warnings.

Further Reading

- Net
  - Kernel Dev mailing list - Best source for the final word. HIGH traffic.
  - Kernel Notes (kt.linuxcare.com) - Weekly essence of kernel dev list.
- Books (Linux Specific)
  - *Understanding the Linux Kernel* by Bovet and Cesati - Most up-to-date of all kernel books (Jan ‘01), tackles the kernel from an architectural standpoint. Good read.
  - *Linux Device Drivers* by Rubini - A bit dated (Feb ‘98), but is the only real HowTo for kernel coding. A must have.
  - *Linux Kernel Internals* by Beck et al. Another architecture book, this one’s a bit too old to be of use (2.0). You’re better off with Bovet.
  - *The Linux Kernel Book* by Card et al. Another 2.0 architecture book.
- Books (All Unix)
  - *The Design of the UNIX Operating System* by Bach. Classic introduction to UNIX design (SVR2).
  - *Unix Internals: The New Frontiers* by Uresh Vahalia. The Best modern UNIX book I’ve seen.
  - *The Design and Implementation of the 4.4 BSD Operating System* by Leffler et al. Another famous book, but I prefer Vahalia for most use.

Q & A