Writing Linux Kernel Modules

Kernel Fibonacci Series Generator

Muli Ben-Yehuda

mulix@mulix.org

IBM Haifa Research Labs

motivation

We already discussed in the last talk why we might want to write a Linux kernel module. Now we are going to learn how, by writing a Linux Kernel Fibonacci Series Generator.

```
mulix@tea:~$ sudo rmmod kfib
mulix@tea:~$ sudo insmod
~/kernel/trees/2.4.22-pre5-kfib/drivers/char/kfib.o
mulix@tea:~$ for i in ‘seq 1 20‘; do cat /dev/fibonacci | tr ’\n’ ’ ’; do
1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946
mulix@tea:~$
```

what are kernel modules?

- kernel modules are object files that contain kernel code
- they can be loaded and removed from the kernel during run time
- they contain unresolved symbols that are linked into the kernel when the module is loaded
- kernel modules can only do some of the things that built-in code can do - they do not have access to internal kernel symbols

module utilities

- modules are loaded via insmod(1) and modprobe(1)
- loaded modules can be listed via lsmod(1)
- modules can be removed via rmmod(1)
- modules cannot be removed while there are references to them. lsmod(1) will show a module's refcount

creating an empty module

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

int init_module(void)
{
        /* do something to initialize the module */

        return 0; /* on error, return -EERR */
}

void cleanup_module(void)
{
        /* cleanup after the module */
}
```

creating a character device

```
int init_module(void)
{
        int ret;

        init_kfib(&fibonacci);

        if ((ret = register_chrdev(KFIB_MAJOR_NUM, "kfib", &kfib_fops)) <
                printk(KERN_ERR "register_chrdev: %d\n", ret);

        return ret;
}

void cleanup_module(void)
{
        unregister_chrdev(KFIB_MAJOR_NUM, "kfib");
}
```

creating a character device, cont'

```
static int kfib_open(struct inode *inode, struct file *filp)
{
        filp->private_data = &fibonacci;

        return 0;
}

static int kfib_release(struct inode *inode, struct file *filp)
{
        return 0;
}

struct file_operations kfib_fops = {
        .owner = THIS_MODULE,
        .open = kfib_open,
        .release = kfib_release,
        .read = kfib_read
};
```

kfib_read

```
static int kfib_read(struct file *filp, char *ubuf, size_t count,
                     loff_t *f_pos)
{
        struct kfib* fib;
        unsigned int term;
        char kbuf[16] = {0,};
        size_t len;

        if (*f_pos != 0) /* second time we're called? we're through */
                return 0;

        fib = filp->private_data;
        if (!fib)
                BUG();

        ...
```

kfib_read, cont'

```
        ...

        spin_lock(&fib->lock);
        term = __kfib_calc_next_term(fib);
        spin_unlock(&fib->lock);

        snprintf(kbuf, sizeof(kbuf) - 1, "%d\n", term);
        kbuf[sizeof(kbuf) - 1] = '\0';
        len = min(count, strlen(kbuf) + 1);
        if (copy_to_user(ubuf, kbuf, len))
                return -EFAULT;

        *f_pos += len;
        return len;
}
```

userspace, kernelspace

When working in kernel memory, it is extremely important to not access userspace pointer directly, since they may be corrupt or malicious (not point where they are supposed to, or point where they are most definitely not supposed to). User memory must always be accessed via copy_to_user() or copy_from_user(). Other access methods, optimized for various scenarios, also exist.

```
if (copy_to_user(ubuf, kbuf, len))
        return -EFAULT;
```

SMP, races and locking

When working in the kernel, one must always pay attention to potential races. For example, what happens if two cpu's execute this code at the same time?

```
static unsigned int __kfib_calc_next_term(struct kfib* fib)
{
        unsigned int term;

        if (!fib)
                BUG();

        term = fib->nmin1 + fib->nmin2;
        fib->nmin2 = fib->nmin1;
        fib->nmin1 = term;

        return term;
}
```

SMP, races and locking, cont'

Always lock to protect against races. The Linux kernel has various kinds of locks, depending on the usage scenarios. In this case we use the simplest one, a spinlock:

```
spin_lock(&fib->lock);

term = __kfib_calc_next_term(fib);

spin_unlock(&fib->lock);
```

The other simple primitive lock type is a mutex. The difference is that a spinlock is much faster, and code holding a spinlock must not sleep (schedule out). Due to kernel 2.4's non-preemptability, spinlocks are a NOP with just one CPU.

building the module

Add an entry to Documentation/Configure.help

```
+Kernel Fibonacci Series Generator
+CONFIG_KERNEL_FIBONACCI
+  This is a driver for a kernel fibonacci series generator.
```

Add an entry to the appropriate Config.in file

```
+tristate 'Kernel Fibonacci Series Generator'CONFIG_KERNEL_FIBONACCI}\
```

Add an entry to the appropriate Makefile

```
+obj-$(CONFIG_KERNEL_FIBONACCI) += kfib.o
```

Now make *config, and don't forget to choose your new kernel fibonacci series generator. 'make modules' and 'make modules_install', and you're all set!