Introduction to Kernel Modules

KERNEL MODULES

What is a kernel module?

A kernel module is code that can be dynamically loaded into the kernel during runtime. This makes the Linux kernel extensible so that not all functionality has to be compiled into the kernel during the initial creation. From our experience making our own system call, we know that compiling a kernel is a lengthy process and that was a SMALL kernel.

The first purpose of modules you might think of is probably device drivers that allow the kernel to interact with a new type of hardware. However, modules can add a host of other functionality like filesystems, network protocols, cryptographic libraries, etc. We will now create a simple hello_world module. There are ways of making a module more robust or more convenient to program, but we will be presenting the simplest module possible.

How to write a hello world kernel module?

step 1: Add an initializing function and an exiting function. A module doesn't use a mandatory “main” function, but instead registers a certain function to initialize variables call any functions that will get the module rolling for whatever its purpose is. When a module is taken out of the kernel, any memory allocated must be freed and perhaps hardware has to be returned a specific state. For example, if a hard drive device driver was taken out of the kernel, the kernel won't be able to interact with the hard drive. Thus it would probably not be a good idea to just leave the hard drive spinning.

The following C file presents a complete working module that prints a kernel message to dmesg. module_init and module_exit are macros that label certain functions as the initial and exiting functions. These macros, along with many other convenient structures and functions, are available from the include/linux/module.h header file from a Linux kernel source tree.

hello_world.c:

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

int init_func(void)
{
    printk("Hello world 1.\n");
    return 0;
}

void cleanup_func(void)
{
    printk("Goodbye world 1.\n");
}

module_init(init_func);
module_exit(cleanup_func);
```
step 2: Of course the most important step is to license your module under the GPL license and claim credit! The module.h file provides a few macros to help us with this as well.

hello_world.c:

```c
#define DRIVER_AUTHOR "Your name <Your@email.org>"
#define DRIVER_DESC   "hello world example module"
MODULE_LICENSE("GPL");
MODULE_AUTHOR(DRIVER_AUTHOR);
MODULE_DESCRIPTION(DRIVER_DESC);
```

How to compile a kernel module?

Compiling a kernel requires the header files of the currently executing kernel. To see which kernel you're using type `uname -r`. If this kernel uses modules, you should be able to locate a directory in `/lib/modules/` that correlates to your kernel's version. The following Makefile adds as the classpath the location of your kernel's header files through a symbolic located in the `/lib/modules/<kernel version>`. `obj-m` indicates a list of kernel modules to build. By including the keyword “modules” on the command line, the compiler knows your code depends on the kernel.

Makefile:

```make
obj-m := hello_world.o
KDIR := /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)

default:
    $(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules
```

Then simply running the “make” command on the command-line will compile the hello_world.c file into a module.

How to load a kernel module?

Normally the `modprobe` command would load a module based on its name, but this requires the module to be installed in the `/lib/modules/<kernel version>` directory. A more explicit method is the `insmod` command. To load your module type the following:

```
# insmod hello_world.ko
```

After loading your module the function registered as the initial function will be run. Try running the `dmesg` command. You should see a long message ending similar to the following:

```
# dmesg
... 
VFS: Mounted root (reiserfs filesystem) readonly. 
Freeing unused kernel memory: 144k freed 
et0: link up 
Hello world!
```
To see the modules currently loaded in the kernel use the `lsmod` command where you should see:

```
# lsmod
Module      Size    Used by
hello_world 1280    0
```

To see the module information you set with the license, author, and description macros use `modinfo`. For my module I see:

```
# modinfo hello_world.ko
filename: hello_world.ko
license: GPL
author: Jeremy Bongio <bongiojp@clarkson.edu>
description: hello world example module
vermagic: 2.6.17.13-VMware mod_unload PENTIUM4 REGPARM gcc-4.1
depends:
```

And to unload the module use the `rmmod` command and notice that dmesg has another message:

```
# rmmod hello_world
# dmesg
... 
Goodbye world!
```

Now you know how to create a simple kernel module. There is much more to creating device drivers and other useful abstractions but this is how they are all created and loaded. Since we're running in kernel space, we have access to all of the kernel memory and, thus, kernel functions and data structures. Let's take a peek at some of those structures by writing a module that looks into the metadata for running tasks.
Task Spy Module

Tasks are represented in kernel by process control blocks, a data structure called task_struct, which is defined in /usr/src/linux/include/linux/sched.h

struct task_struct {
    volatile long state;
    unsigned int flags;
    int prio, static_prio, normal_prio;
    ...
    struct list_head tasks;
    struct mm_struct *mm, *active_mm;
    ...
    pid_t pid;
    pid_t tgpid;
    cputime_t utime, stime, utimescaled, stimescaled;
    ...
}

Reference:
http://lxr.linux.no/linux+v2.6.17/include/linux/sched.h#L696

Let's write a module that prints the command and process IDs of tasks running on the system.

#include <linux/module.h>
#include <linux/sched.h>
#include <linux/list.h>

void print_data_for_process(struct task_struct *my_current) {
    struct task_struct *next_process = NULL;
    struct list_head *head_of_children_list = &(my_current->children);

    //////////////////////////////////////////////////////////////////////////
    /* PLACE YOUR CODE FOR ACCESSING AND PRINTING DATA HERE */
    printk("%s(%d)\t\n", my_current->comm, my_current->pid);
    //////////////////////////////////////////////////////////////////////////

    // recursively call this function on each child of the current process
    list_for_each_entry(next_process, head_of_children_list, sibling) {
        print_data_for_process(next_process);
    }
}
int __init init_func(void) {
    struct task_struct *my_current;

    printk("process_monitor: printing stuff\n");

    // We get the "current" variable from sched.h
    // "current" is the current process running now, but we use it
    // to follow the links of parents to the very first process.
    my_current = current;

    // The first process is the "swapper". It's initialized by a macro
    // defined in include/linux/init_task.h. Check it out.
    while (my_current->pid != 0) {
        my_current = my_current->parent;
        printk("%d\n", my_current->pid);
    }

    printk("Process information...\n");

    // Call a function to recurse through all children of the first
    // process and print their associated data.
    print_data_for_process(my_current);

    return 0;
}

void __exit cleanup_func(void) {
    printk("Going down for reboot ... \n\n\njust kidding!\n");
}

module_init(init_func);
module_exit(cleanup_func);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Your Name <email at address>");
MODULE_DESCRIPTION("process monitoring module");

Let's compile our kernel module using a Makefile similar to our Hello World example, insmod it, and
run dmesg to see the data it prints out.

Recall when we added a system call to the kernel by modifying kernel source code and recompiling.
Do you think it would be possible to add a system call to the system call table dynamically, through a
module, without requiring kernel recompilation? It is not possible to do this, or at least not easily possible—why do you think this is the case? The sys_call_table is of fixed size, based upon a #define for __NR_syscall_max, so new entries cannot be added to the system call table without overwriting a const declaration, but existing ones could be overridden by your own function!

(In arch/x86/kernel/syscall.c (or syscall_32.c) we see it defined as such:
const sys_call_ptr_t sys_call_table[__NR_syscall_max+1] = { ...}

**Task Spy Module Using ProcFS**

Parsing the kernel buffer every time we want to view data from our module can be annoying, since that buffer is shared, and it's possible for multiple things to print to the buffer in whatever order they are scheduled. Furthermore, the kernel buffer is really meant to be read-only for viewing status messages, so now we will create a similar module that uses procfs instead of the kernel buffer.

```c
#include <linux/module.h>
#include <linux/proc_fs.h>
#include <linux/seq_file.h>
#include <linux/sched.h>
#include <linux/list.h>
#include <linux/mm.h>

struct proc_dir_entry* dir;
struct proc_dir_entry* entry;

/*
 * Prints information about a process to a sequential procfs
 * file. Called by the sequential file's show() method.
 *
 * This is the function you should work within.
 */
void print_data_for_process(struct seq_file *m, struct task_struct *my_current) {
    struct task_struct *next_process = NULL;
    struct list_head *head_of_children_list;

    
    
    
```
static void *process_mon_start(struct seq_file *m, loff_t * pos)
{
    printk("process_monitor: starting to print\n");
    if ( !( *pos ) )
        return SEQ_START_TOKEN; // void* v parameter
    else
        return NULL; //nothing more to write.
}

static void *process_mon_next(struct seq_file *m, void *v, loff_t * pos)
{
    return NULL;
}

static void process_mon_stop(struct seq_file *m, void *v)
{
    printk("process_monitor: all done printing\n");
}

static int process_mon_show(struct seq_file *m, void *v)
{
    struct task_struct *my_current;
    printk("process_monitor: printing stuff\n");

    // We get the "current" variable from sched.h
    // "current" is the current process running right now but we use it
    // to follow the links of parents to the very first process.
    my_current = current;

    // The first process is the "swapper". It's initialized by a macro
    // defined in include/linux/init_task.h. Check it out.
    while (my_current->pid != 0) {
        my_current = my_current->parent;
        // seq_printf(m, "%d\n", my_current->pid);
    }

    // Print a header for our proc file.
    seq_printf(m, "Process\tet cetera ...
");

    // Call a function to recurse through all children of the first
    // process and print their associated data.
    print_data_for_process(m, my_current);
    return 0;
}
static struct seq_operations process_mon_ops = {
    .start = process_mon_start,
    .next = process_mon_next,
    .stop = process_mon_stop,
    .show = process_mon_show
};

/*
 * Called when a proc file is "opened" with the open system call.
 * When reading the proc file we created, this is the starting point.
 * Open defines the functions that seq_read and seq_lseek should use
 * to put data together.
 */
static int process_mon_open(struct inode *inode, struct file *file)
{
    return seq_open(file, &process_mon_ops);
}

/*
 * Each of the function pointers in this data struct correlate to a
 * system call. file_operations->open is called when the open system
 * call is used on a proc sequential file. file_operations->seq_read
 * is called when the read system call is used. And so on.
 */
static const struct file_operations file_operations = {
    .open = process_mon_open,
    .read = seq_read,  // already made by seqfile code
    .llseek = seq_lseek, // already made by seqfile code
    .release = seq_release // already made by seqfile code
};

/*
 * Module initialization function.
 * Creates a proc directory to hold our data file.
 * Creates a data file and associates the file_operations struct
 * to be the functions used to access the file from userspace.
 */
int __init init_func(void)
{
    dir = proc_mkdir("ourprocessmon", NULL);
    if (!dir) {
        printk("<1>process_monitor: Could not create procfs directory.
";
        return -1;
    }
    entry = create_proc_entry("processes", 0755, dir);
    if (!entry) {

printk("<1>process_monitor: Could not create procfs file.\n");
return -1;
}
entry->proc_fops = &file_operations;
entry->owner = THIS_MODULE;

printk("process monitor ... INITIALIZED!!!\n");
return 0;
}

void __exit cleanup_func(void) {
    printk("Going down for reboot ... \n\n\n just kidding!\n");
}

module_init(init_func);
module_exit(cleanup_func);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Your Name <email at address>"ateway);
MODULE_DESCRIPTION("procfs process monitoring module");

A few functions are used together to form sequential data in procfs, but, for our purpose, we don't necessarily need to use them all. Let's take a look at a few procfs-related functions we implement in this module:

First, process_mon_ops->start() is the first function to be called when the read() system call is triggered. Its purpose is to initialize any data needed and return that data, which becomes an argument to the rest of the functions. In our case it's easier to loop in the show() function rather than start() passing a pointer to the information. "pos" represents how many iterations we have been through so far (from functions like start() to show() to next() to stop() and then back again for another piece of data).

Second, process_mon_ops->show() is called to actually show whatever data it was told to use by either the next() function or start() function.

Third, process_mon_ops->next() is called if show() returned non-zero. Next will use the "v" data structure to find the next piece of information to print.

Fourth, process_mon_ops->stop() is called to free any dynamically allocated data and return.

At this point, we need only concern ourselves with show():

static int process_mon_show(struct seq_file *m, void *v) {
    struct task_struct *my_current;
    printk("process_monitor: printing stuff\n");
We get the "current" variable from sched.h
"current" is the current process running right now but we use it
to follow the links of parents to the very first process.
my_current = current;

The first process is the "swapper". It's initialized by a macro
defined in include/linux/init_task.h. Check it out.
while (my_current->pid != 0) {
    my_current = my_current->parent;
    // seq_printf(m, "%d\n", my_current->pid);
}

Print a header for our proc file.
seq_printf(m, "Process\tet cetera ...\n");

Call a function to recurse through all children of the first
process and print their associated data.
print_data_for_process(m, my_current);
return 0;
}

So, the show() function locates the root of the list, passes that to the print_data_for_process() function,
and then the print_data_for_process() function, which recursively calls itself over each-next task in the
list. It is almost the same as in the last module we just wrote, but the main difference is that the printk()
function call has been replaced with a seq_printf() function call, which prints data into our virtual file
in procfs:

seq_printf(m, "%s(%d)\t\n", my_current->comm, my_current->pid);

Lastly, before compiling and inserting this module, let's take a look at the file_operations struct:

static const struct file_operations file_operations = {
    .open = process_mon_open,
    .read = seq_read, // already made by seqfile code
    .llseek = seq_lseek, // already made by seqfile code
    .release = seq_release // already made by seqfile code
};

This data structure should look familiar—here, we are setting the backend functions that will handle
file open, read, seek, and release functionality when an application (such as the cat command) attempts
to read from our virtual file in procfs.

If we compile and insmod this module, then we should see an entry in the /proc directory, through
which we can access information about current tasks.
**Procedure**

Refer to the sample Makefile for compiling each kernel module:

1. (In class) Make a hello_world.c kernel module (according to the first example), which simply has your name set as the author and uses printk(..) to announce to the kernel buffer when it is loaded and unloaded

2. (In class) Test the tasks.c kernel module (according to the second example)

3. (In class) Test the procfs_tasks.c kernel module (according to the third example)

4. In procfs_tasks.c, modify the print_data_for_process(...) function: Edit the seq_printf(...) to not only print task command names (->comm) and process IDs (->pid), but to print at least one other variable in the task_struct ( http://lxr.linux.no/linux+v2.6.17/include/linux/sched.h#L696 )

5. Similar to the template hello_world.c, write your own new module called goodbye_world.c: Have this module do something “bad” like alter task_struct (see tasks.c) or simply perform an infinite while loop

6. For fun, optionally check out Kprobes by reading /usr/src/linux/Documentation/kprobes.txt

**What to Submit**

scp the following files to USER@polaris.clarkson.edu:/afs/cu/class/cs444/sp12/students/USER/module/

- hello_world.c
- tasks.c
- procfs_tasks.c (your modified version)
- goodbye_world.c (your own module)
- README (briefly explaining what you observed when loading your goodbye_world.c module)

**References**

Linux Kernel Modules

http://www.captain.at/programming/kernel-2.6/
http://www.faqs.org/docs/kernel/x41.html

This document was updated by Patrick F. Wilbur, based upon materials originally by Jeremy Bongio and Wenjin Hu