Worksheet 7

Unix Systems Programming (COMP2002)
Linux Kernel Modules
Updated: 27/04/2021

Note

This worksheet is designed around a particular operating environment. You will need:
* A Linux environment with kernel version 4.13 or greater.
* Access to 'sudo'
* Linux headers installed (See: Compiling a Kernel Module for more details)

1 Research

Research the following and make note of what they are used for:

a. lsmod
b. insmod
c. rmmod
d. printk()
e. struct sk_buff
f. tail -f
g. /var/log/syslog
h. dmesg

2 Creating a Kernel Module

We will be creating a simple kernel module that will be printing 'Hello World!' to the kernel log. Each kernel module will at least have the following functions:

- An init function

    - Called whenever the module is installed
- An exit function
    - Called whenever the module is removed

Create a file named **hellokernel.c**. We will require three header files here:

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/init.h> // This is typically built-in
```

Next, we will need to create our init and exit functions. The name (as of Linux 2.4) may be anything you want, but it is good practice to make it clear what they are. Take note of the special macros '___init' and '___exit':

```
static int __init hello_init()
{
    // Use printk() to create your modules's 'Hello World!'
}

static void __exit hello_exit()
{
    // ...
}
```

Finally, we need to register our functions in order to designate what exactly they are:

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

3 Compiling a Kernel Module

Compiling the kernel module is somewhat different from your usual program. Use the following makefile to compile your module:

```
obj-m=hellokernel.o

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

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

Note

If the Linux headers are not installed in your system, the compilation will fail here. Run the following command to install them (for Ubuntu):
sudo apt-get install build-essential linux-headers-$(uname -r)

4 Running Your Module

Consider the topics researched in part one. Now that you have a compiled kernel module, how will you install it? How will you check to see if it worked? How will you remove it?

Note

You will need sudo access to do this!

5 Packet Sniffing Module

We will next extend our Linux Kernel Module to operate on network traffic using the netfilter project. Much of this is fairly prescriptive in nature, but it's highly encouraged to approach this with a curiosity for customisation.

Your module will need more header files to accommodate the extended functionality:

```
#include <linux/ip.h>
#include <linux/netfilter.h>
#include <linux/netfilter_ipv4.h>
#include <linux/skbuff.h>
#include <linux/netdevice.h>
#include "/usr/include/linux/if_ether.h"
```

Next, we will be utilising the struct nf_hook_ops for registering our 'hooks' into the network stack – one for incoming and one for outgoing. For ease, we will make these structs static variables outside of any function, **likely just underneath your header file includes**:

```
static struct nf_hook_ops netfilter_ops_in;
static struct nf_hook_ops netfilter_ops_out;
```

The 'hook' that will be used will itself be a function that we write. In this case, we will be using the same function for both – but you may write different functions if you so wish. This 'hook' will be called whenever we send or receive a packet.

```
unsigned int main_hook(void* priv,
                       struct sk_buff* skb,
                       const struct nf_hook_state* state)
{
    printk(KERN_INFO "packet received\n");
    return NF_ACCEPT; // NF_ACCEPT allows packets, NF_DROP rejects them
}
```

Inside your *hello_init()* function, we will now add code to configure our *nf_hook_ops* structs and register our hooks accordingly:

```
netfilter_ops_in.hook = main_hook; // What function to call?
netfilter_ops_in.pf = PF_INET; // Protocol Family
netfilter_ops_in.hooknum = NF_INET_PRE_ROUTING; // Incoming traffic
netfilter_ops_in.priority = NF_IP_PRI_FIRST; // What priority is this hook?

netfilter_ops_out.hook = main_hook;
netfilter_ops_out.pf = PF_INET;
netfilter_ops_out.hooknum = NF_INET_POST_ROUTING; // Outgoing traffic
netfilter_ops_out.priority = NF_IP_PRI_FIRST;

nf_register_net_hook(&init_net, &netfilter_ops_in); // Register the incoming traffic hook
nf_register_net_hook(&init_net, &netfilter_ops_out); // Register the outgoing traffic hook
```

Similarly, you will want to edit your *hello_exit()* function to deregister our hooks:

```
nf_unregister_net_hook(&init_net, &netfilter_ops_in);
nf_unregister_net_hook(&init_net, &netfilter_ops_out);
```

Compile your module, run it, and see what happens when using a web browser.

6 Expanding On What You've Learned

The above example is relatively simple compared to what is possible. The sk_buff structure contains a great deal of information that you may use to expand upon what is output by your module – you are even able to implement logic for determining when to accept and when to drop a packet based on the packet that has been received.

Try using the sk_buff struct to expand on your kernel module. Are you able to determine the IP addresses associated witht he packet?

There are any number of possibilities here! For inspiration, you might be interested in ideas such as Singwall, the singing firewall that audibly chirps based on the incoming packet's port.