LAB 7: KERNEL MODULE & CHARACTER DRIVER

Dr. Kazi Md. Rokibul Alam & JAKARIA
Modified by: Abdul Aziz

KERNEL MODULE

The kernel is the core of any operating system and is responsible for managing system resources. Broadly, the Linux kernel can be of two types.

Monolithic kernels: This is a single executable file in which all the modules are part of the kernel. In order to add anything to the existing kernel, developers have to rebuild the complete kernel and add the new functions.

Modular kernels: Modular kernels provide developers an option to add new functionality to the existing kernel by plugging the new code, also known as ‘**modules**’ at run time.

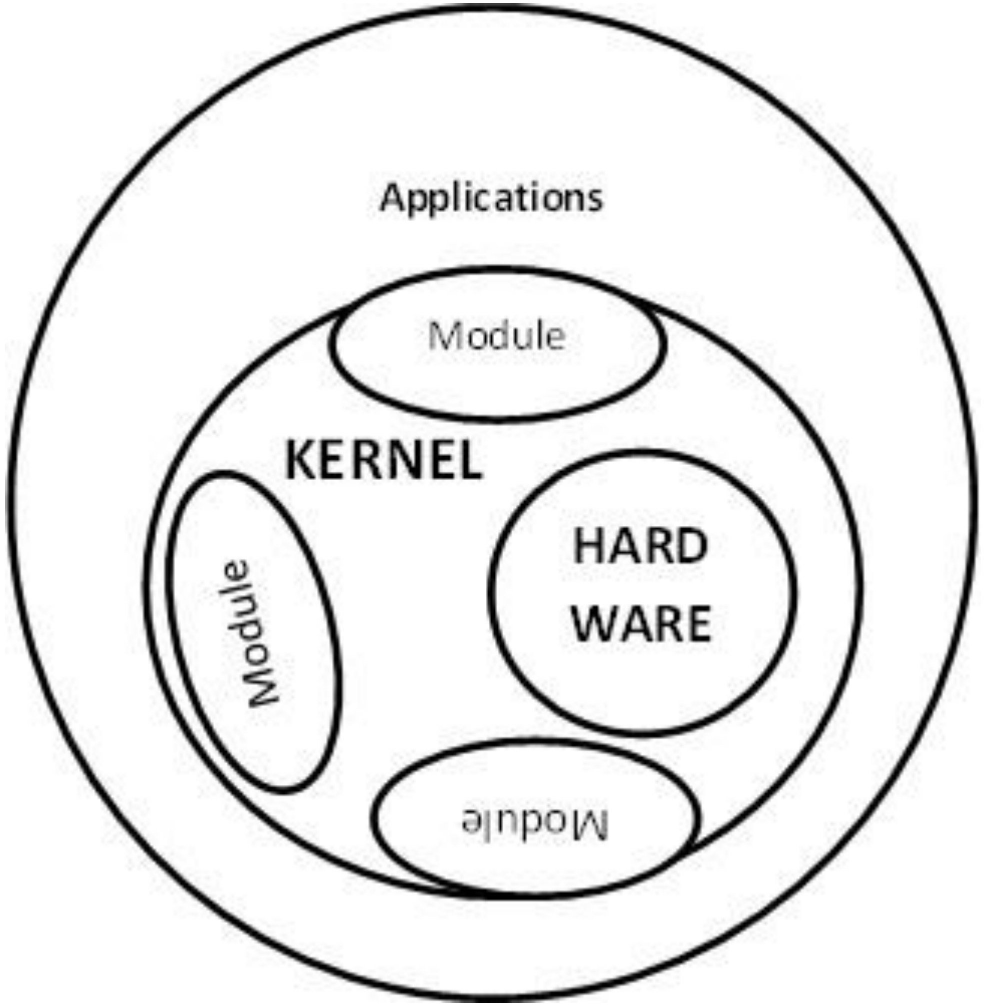
KERNEL MODULE

What are kernel modules?

- Kernel modules are pieces of code, which can be loaded and unloaded from a kernel, on demand. A Linux Kernel Module (**LKM**) can be added at run time without even requiring a reboot or even a rebuild of the running kernel. The LKM will have **a.*ko*** extension.
- The **LKM** will act as the interface between a user space application and the Linux kernel. Any request to access the hardware from an application goes via the **LKM** to the kernel, and then to the actual hardware (see Figure in the next slide).
- To know the list of modules running in a Linux kernel you can use the '**lsmod**' command, which actually gives the list of running modules at that point of time, by reading ***'/proc/modules'*** as shown in Figure

KERNEL MODULE

```
[root@Ashish-LFY-Article helloworld]# lsmod
Module                  Size  Used by
ipmi_si                43275  1
ip6table_filter         2855  0
ip6_tables             19296  1 ip6table_filter
ebtable_nat             2039  0
ebtables               18401  1 ebtable_nat
ipt_MASQUERADE          2400  3
iptable_nat             6156  1
nf_nat                 23292  2 ipt_MASQUERADE,iptable_nat
nf_conntrack_ipv4       9440  4 iptable_nat,nf_nat
nf_defrag_ipv4          1449  1 nf_conntrack_ipv4
```



KERNEL MODULE

Kernel module management commands:

insmod <module-name>: This command is to insert the new module into the kernel

lsmod**:** This lists the modules that are currently loaded in the kernel

modinfo <module-name>: This is to get complete information about the module

rmmod <module-name>**:** This command is to remove the module from the kernel

modprobe <module-name>: This works the same as insmod but it uses ‘Module Stacking’ to load any module that is required to load the current module.

modprobe r <module>: To remove the module from the kernel

KERNEL MODULE

```
[root@Ashish-LFY-Article helloworld]# insmod helloworld.ko
[root@Ashish-LFY-Article helloworld]#
[root@Ashish-LFY-Article helloworld]# lsmod | grep -i helloworld
helloworld              1360  0
[root@Ashish-LFY-Article helloworld]#
[root@Ashish-LFY-Article helloworld]# modinfo helloworld.ko
filename:       helloworld.ko
version:        0.1
description:    Sample Module
author:         ASHISH BUNKAR
license:        GPL
srcversion:     7E0D036351D3DDCFF5ABFFE
depends:
vermagic:       2.6.32-131.0.15.el6.x86_64.debug SMP mod_unload modversions
parm:           hello:int
[root@Ashish-LFY-Article helloworld]#
[root@Ashish-LFY-Article helloworld]# rmmod helloworld.ko
[root@Ashish-LFY-Article helloworld]#
[root@Ashish-LFY-Article helloworld]# lsmod | grep -i helloworld
[root@Ashish-LFY-Article helloworld]#
```

KERNEL MODULE

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

int hello_init(void){
    printk(KERN_ALERT "TEST: Hello world\n");
        return 0;
}
void hello_exit(void){
    printk(KERN_ALERT "TEST: Good bye\n");
}
module_init(hello_init);
module_exit(hello_exit);
```

KERNEL MODULE

hello_init(): This is called when the module is inserted into the kernel using **insmod**. This function gets invoked by the '**module_init**' macro. The **init** function is responsible for registering the module with the kernel.

hello_exit(): This function is called when the module is removed from the kernel using **rmmod**. This function gets invoked by the '**module_exit**' macro. This function removes and cleans up the inserted module.

KERNEL MODULE

- Macros **module_init** (**hello_init**) & **Module_init** (**hello_exit**): Using these macros, programmers can give user defined names to the init and cleanup functions. These macros are defined in <**linux/init.h**>.
- **Printk**: In kernel module programming, '**printk**' is used to print kernel messages in to the kernel logs. **Printk** messages are linked to the priority associated with them. For all behavioural purposes, we use '**printk**' in kernel module programming much as we use '**printf**' in user level C programs.

KERNEL MODULE

MAKE FILE:

```
obj-m += Hello.o

KDIR = /usr/src/linux-headers-2.6.32-21-generic

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

KERNEL MODULE

- **Compiling and building the module:** Use a **makefile** to compile and build the sample hello module.
- Use the '**make**' command to compile and build the **hello** kernel module program.
- Once the module is compiled and built using make, the 'module.ko (**hello.ko**)' will be created.
- Now that we have the **hello.ko** file, insert this module into or remove it from the kernel by using the **insmod/rmmod** commands.
- Use **dmesg** to see the output.

KERNEL MODULE

MAKE file explanation

obj-m += Hello.o

This tells kbuild that there is one object in that directory, named **Hello.o**. **Hello.o** will be built from **Hello.c** or **Hello.s**.

- **obj-m** is a list of what kernel modules to build. The .o and other objects will be automatically built from the corresponding .c file (no need to list the source files explicitly).

KDIR = /usr/src/linux-headers-2.6.32-21-generic

- Directory of the working linux kernel. You can find it by writing: **uname –r. You have to change it or use $(shell uname –r) instead of '2.6.32-21-generic'**

KERNEL MODULE

- **How can you add several object file? Check the syntax.**

all:

$(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules

- You have a **Makefile** that calls make with the -**C** option to change directory to where your kernel source is.
- Make then reads the **Makefile** there (in the kernel source dir). **SUBDIRS** is where your module source code is. Them ‘**modules’** say to make module**.**
- **So, create a directory of any named and place hello.c and makefile there**.
- The kernel build system will look for the **Makefile** in your module's

KERNEL MODULE

```
#include <linux/init.h>
#include <linux/module.h>
#include
<linux/moduleparam.h>
int param_var =0;
```

KERNEL MODULE

Passing Command Line Arguments to a Module

- Modules can take command line arguments, but not with the **argc/argv** you might be used to.
- To allow arguments to be passed to your module, declare the variables that will take the values of the command line arguments as global and then use the **module_param**() macro, (defined in **linux/moduleparam.h**) to set the mechanism up. At runtime, **insmod** will fill the variables with any command line arguments that are given, like **insmod hello.ko param_var=5.**

KERNEL MODULE

- The variable declarations and macros should be placed at the beginning of the module for clarity. The example code should clear up my admittedly lousy explanation.
- **The module_param() macro takes 3 arguments**: the **name** of the variable, its **type** and **permissions** for the corresponding file in sysfs. Integer types can be signed as usual or unsigned. If you'd like to use arrays of integers or strings see **module_param_array**() and **module_param_string**().

Summary Commands

- Go to the directory(In Root) where your **.c** and **make** files are existing by **cd** command.
- Then serially execute the following commands:
- **make**
- **insmod hello.ko**
- **rmmod hello.ko**
- **Tail –f /var/log/syslog**
- Type the **last command** in a **new terminal** window (ctrl+alt+t) to see the **log** message

MAKE file Revisited

- We walked a long way on the path of linux device driver programming.
- On the horizon we can now see our first device driver. So, little work through from here is needed. Hold your attention ☺
- Following lines are added to your make file (In this folder we have provided the new make file with this line):

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

- If you change your **.c** file then new build is necessary by **make** command.

MAKE file Revisited

- Write the following command
- **make clean**
- This will clean your **.ko** other files created after build. Then rebuild by writing **make** command again

Necessary Commands

- All explanation is provided in the **web page** and **6th video** of the **video series** that we have provided. Summary of steps is given below:
- **make clean**
- **make**
- **insmod Driver.ko**
- **mknod /dev/memory c 240 0** [Device file] [To see go to **/dev**]
- **chmod 666 /dev/memory** [Permission]
- **echo -n abcdef >/dev/memory** [Write]
- **cat /dev/memory** [Read]

Write function

```
ssize_t hello_write(struct file *pfile, const char __user *buffer, size_t
length, loff_t *offset)
{
	int nbytes;
	nbytes = length - copy_from_user(mybuffer + *offset, buffer,
length);
	*offset += nbytes;

   return nbytes;
}
```

Write function

- **echo -n abcdef >/dev/memory**
- After this command **write function** invoked
- This function has some change from the web page.

nbytes = length - copy_from_user (mybuffer + *offset, buffer, length);

- Here length = 6 bytes (‘**a’b’c’d’e’f’)**
- *offset = 0 for first call.
- **mybuffer** is in kernel space. So, we write 6 bytes data to **mybuffer** from user **buffer**
- copy_from_user (mybuffer + *offset, buffer, length) -> returns number of bytes not written.

Write function

- nbytes = length - copy_from_user (mybuffer + *offset, buffer, length);
- So, **nbytes** = 6 – 0 = 6
- *offset += nbytes;

-> Offset = 6

- return nbytes;

-> Return 6

Read function

```
ssize_t hello_read(struct file *pfile, char __user *buffer, size_t length,
loff_t *offset)
{       int nbytes;
        int maxbytes;
        int bytes_to_do;

        maxbytes = 6 - *offset;
        if(maxbytes > length)
                bytes_to_do = length;
        else
                bytes_to_do = maxbytes;

        nbytes = bytes_to_do - copy_to_user(buffer, mybuffer + *offset
```

Read function

- This function has some change from the web page.
- Here **length** is the length of user buffer where we move data from kernel.
- **For first time call:**
- maxbytes = 6 - *offset; [here first time *offset = 0]
- Here 6 means we want to read six byte. You can put any value.
- if(maxbytes > length) [maxbytes = 6]

```
                bytes_to_do = length;
        else
                bytes_to_do = maxbytes; [=6]
```

Read function

nbytes = bytes_to_do - copy_to_user(buffer, mybuffer + *offset, bytes_to_do);

- So, **nbytes** = 6 – 0 = 6
- *offset += nbytes;

-> Offset = 6

- return nbytes;

-> Return 6

- This function terminate when it returns 0;
- So, again this function is called and this time *offset = 6

Read function

nbytes = bytes_to_do - copy_to_user(buffer, mybuffer + *offset, bytes_to_do);

- So, **nbytes** = 0 – 0 = 0
- *offset += nbytes;

-> Offset = 6

- return nbytes;

-> Return 0

- This function terminate when it returns 0;
- So, all data are transferred.

KMALLOC

IMPORTANT:

```
mybuffer=kmalloc(1,GFP_KERNEL);
memset(mybuffer,'\0',1);
```

Here 1 is page size. Your page size can be of 32 or 64 byte. Commands to see:

Getconf PAGESIZE or

Getconf PAGE_SIZE

Summary

Vary memory allocation bytes and number of bytes to transfer in the code. Then you can realize clearly. ☺

Attention Please ☺

- You must practice these workflow for proper understanding. Use **files** and **videos.**
- Otherwise you can not give answers to your VIVA questions.
- Thank you . ☺