TRƯỜNG ĐẠI HỌC BÁCH KHOA – ĐẠI HỌC QUỐC GIA TP.HCM
TRUNG TÂM KỸ THUẬT ĐIỆN TOÁN
268 Lý Thường Kiệt, Phường 14, Quận 10, TP.Hồ Chí Minh
Điện thoại: 84-8-3864 7256 ext 5371 – Fax: 84-8-3865 8687
Website: **www.cce.hcmut.edu.vn**
E-mail: **dientoan@hcmut.edu.vn**

Lab 10
Linux Driver

MSc.Hoàng Lê Hải Thanh
High Performance Computing Laboratory, HCMUT, VNU-HCMC
thanhhoang@hcmut.edu.vn

I. GOALS

- Review Linux Driver concept

II. OBJECTIVES

- Learn how to create and load a Linux kernel module
- Know how to inspect Linux installed driver
- Implement a simple Linux Character Device driver

III. PREREQUISITES

- Install the Linux Virtual Machine on your computer. WSL has not provided the kernel header library to compile the kernel module

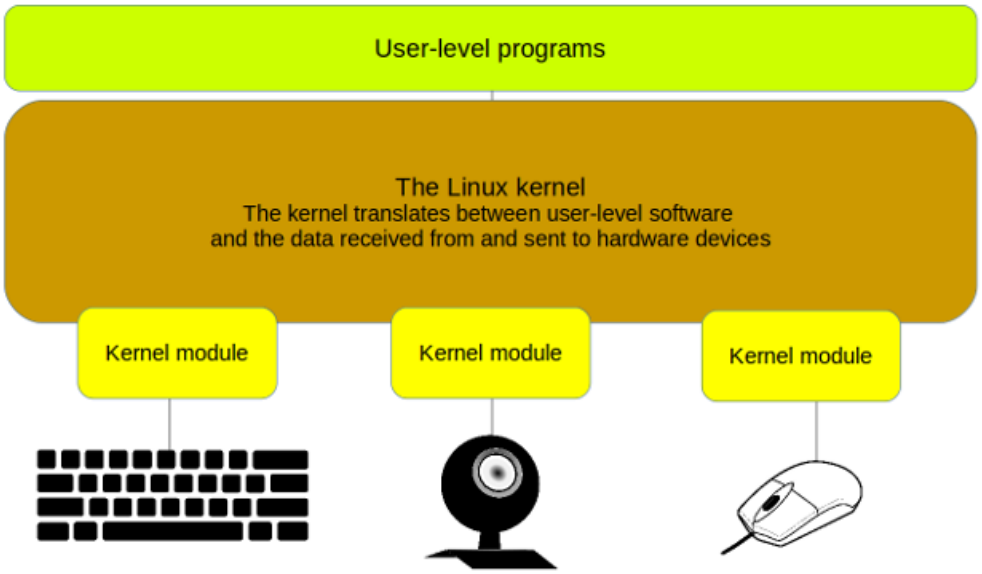
IV. LINUX KERNEL MODULE

1. Introduction to Kernel Module

The kernel is the core component of an operating system. It manages the system's resources, and it is a bridge between your computer's hardware and software.

A monolithic kernel, as used in Linux, though faster than a microkernel, has the disadvantage of lack of modularity and extensibility. On modern monolithic kernels, this has been solved by using *kernel modules*.

A kernel module (or loadable kernel mode) is an object file that contains code that can extend the kernel functionality at runtime (it is loaded as needed); They extend the functionality of the kernel without the need to reboot the system. When a kernel module is no longer needed, it can be unloaded. Most of the device drivers are used in the form of kernel modules.



First, let's look at all currently loaded kernel modules on your machine using the `lsmod` command. What the command does is that it reads `/proc/modules` and displays the file contents in a nicely formatted list:

```
$lsmod
Module                  Size  Used by
binfmt_misc            24576  1
vmw_vsock_vmci_transport    32768  1
vsock                  36864  2 vmw_vsock_vmci_transport
dm_multipath           32768  0
scsi_dh_rdac           16384  0
scsi_dh_emc            16384  0
scsi_dh_alua           20480  0
intel_rapl_msr         20480
...
```

Each line has three columns:

- **Module** - The first column shows the name of the module.
- **Size** - The second column shows the size of the module in bytes.
- **Used by** - The third column shows a number that indicates how many instances of the module are currently used. A value of zero means that the module is not used. The comma-separated list after the number shows what is using the module.

We can discover even more information about a specific kernel module using the `modinfo` command. For example, we can view more information about the `vsock` module, with the following command:

```
$ modinfo vsock
filename:       /lib/modules/5.4.0-109-generic/kernel/net/vmw_vsock/vsock.ko
license:        GPL v2
version:        1.0.2.0-k
description:    VMware Virtual Socket Family
author:         VMware, Inc.
srcversion:     81017F75D1DE6169835D04E
depends:
retpoline:      Y
intree:         Y
name:           vsock
vermagic:       5.4.0-109-generic SMP mod_unload modversions
sig_id:         PKCS#7
signer:         Build time autogenerated kernel key
sig_key:        2E:88:51:F0:25:CE:CD:BB:F2:FC:76:4E:F5:71:F2:A4:3F:3E:DC:F3
sig_hashalgo:   sha512
...
```

The output includes:

- **filename** - the specific path housing the module
- **alias** - the module alias used within the kernel.
- **author** - the author of the module.
- **license** - the module license.
- **srcversion** - the specific version of the module source.
- **depends** - any dependencies a module might have.
- **retpoline** - if the module is retpoline enabled.
- **intree** - if the module is maintained in the kernel Git repository
- **name** - the name of the module.
- **vermagic** - the version of the kernel module.
- **sig_id, signer, sig_key, sig_hashalgo,** and **signature** - all display information about the module's key signature.

2. A Simple Kernel Module

a. Install the Linux Kernel Headers

Before starting, we need to install the corresponding Linux Kernel Headers. Kernel Headers contain the C header files for the Linux kernel, which offers the various function and structure definitions required when compiling any code that interfaces with the kernel, such as kernel modules or device drivers and some user programs.

```
$~ sudo apt-get update
$~ sudo apt-get install build-essential linux-headers-$(uname -r)
```

Next, check if the matching kernel headers have been installed on your system using the following command

```
$~ ls -l /usr/src/linux-headers-$(uname -r)
total 1612
lrwxrwxrwx 1 root root      40 Apr  8  2022 Documentation ->
../linux-headers-5.4.0-109/Documentation
```

```
lrwxrwxrwx 1 root root      33 Apr  8  2022 Kbuild -> ../linux-headers-5.4.0-109/Kbuild
lrwxrwxrwx 1 root root      34 Apr  8  2022 Kconfig -> ../linux-headers-5.4.0-109/Kconfig
lrwxrwxrwx 1 root root      35 Apr  8  2022 Makefile -> ../linux-headers-5.4.0-109/Makefile
-rw-r--r-- 1 root root 1620578 Apr  8  2022 Module.symvers
drwxr-xr-x 3 root root    4096 May  6  2022 arch
lrwxrwxrwx 1 root root      32 Apr  8  2022 block -> ../linux-headers-5.4.0-109/block
...
```

b. A Hello World Kernel Module

We'll start with a conventional “Hello World” program (hello-world.c) that demonstrates the different aspects of the basics of writing a kernel module.

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

/* Your module information here */
MODULE_LICENSE(“GPL”);
MODULE_AUTHOR(“Your Name Here”);
MODULE_DESCRIPTION(“A simple example Hello World module.”);
MODULE_VERSION(“0.01”);

static int __init lkm_example_init(void) {
 printk(KERN_INFO “Hello, World!\n”);
 return 0;
}

static void __exit lkm_example_exit(void) {
 printk(KERN_INFO “Goodbye, World!\n”);
}

module_init(lkm_example_init);
module_exit(lkm_example_exit);
```

Any kernel module must have at least two functions: a "*start*" (initialization) function called **`lkm_example_init`** which is called when the module is loaded into the kernel, and an "end" (cleanup) function called **`lkm_example_exit`** which is called just before it is removed.

Typically, the **`init_module`** function either registers a handler for something with the kernel or replaces one of the kernel’s functions with its own code (usually code to do something and then call the original function). The **`cleanup_module`** function is supposed to undo whatever **`init_module`** did, so the module can be unloaded safely.

At the end of the file, we call **`module_init`** and **`module_exit`** to tell the kernel which functions are loading and unloading functions. This gives us the freedom to name the functions whatever name we like.

Next, we will define some macros for the Licensing (`MODULE_LICENSE`) and Module Documentation (`MODULE_AUTHOR`,..etc) They're simply for documentation and can be viewed when we inspect the module.

We can also see the **`printk`** function. `printk` is one of the most widely known functions in the Linux kernel. It's the standard tool we have for printing messages and usually the most basic way of tracing and debugging. **`printk`** messages can specify a log level as we can see in the example.

Lastly, every kernel module needs to include `linux/module.h`. We might need to include the `linux/kernel.h` only for the macro expansion for the `printk` log level.

c. Compile a Kernel Module

A kernel module needs to be compiled a bit differently from regular userspace programs as we have done in previous sessions. Former kernel versions required us to care much about these settings, which are usually stored in Makefiles. Although hierarchically organized, many redundant settings accumulated in sublevel Makefiles and made them large and rather difficult to maintain.

Fortunately, there is a new way of doing these things, called ***kbuild***, and the build process for external loadable modules is now fully integrated into the standard kernel build mechanism. So, let's look at a simple Makefile for compiling our previous module named hello-world.c:

```
obj-m += hello-world.o

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

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

Now we can compile the module by issuing the command **make**. We will obtain an output that resembles the following:

```
$~  make
make -C /lib/modules/5.4.0-109-generic/build M=/home/hpcc/drivers modules
make[1]: Entering directory '/usr/src/linux-headers-5.4.0-109-generic'
  CC [M]  /home/hpcc/drivers/hello-world.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC [M]  /home/hpcc/drivers/hello-world.mod.o
  LD [M]  /home/hpcc/drivers/hello-world.ko
make[1]: Leaving directory '/usr/src/linux-headers-5.4.0-109-generic'
```

Now we can inspect our kernel module:

```
$~ modinfo hello-world.ko
modinfo hello-world.ko
```

```
filename:       /home/hpcc/drivers/hello-world.ko
version:        0.01
description:    A simple example Hello World module.
author:         Your Name Here
license:        GPL
srcversion:     63B67579C077793FD1F74C0
depends:
retpoline:      Y
name:           hello_world
vermagic:       5.4.0-109-generic SMP mod_unload modversions
```

3. Loading/unloading a kernel module

Now it's time to test our built kernel module. To load a kernel module, use the `insmod` utility. This utility receives as a parameter the path to the *.ko file in which the module was compiled and linked.

```
$~ sudo insmod hello-world.ko
```

If all goes well, you won't see a thing. The printk function doesn't output to the console but rather the kernel log. To see that, we'll need to run:

```
$~ sudo dmesg
...
[ 4867.502807] Hello, World!
```

We can also check to see if the module is still loaded:

```
$ lsmod
Module                  Size  Used by
hello_world            16384  0
...
```

Question: Use **`modinfo`** to inspect out module.

Finally, we can unload the module from the kernel is done using the rmmod command, which receives the module name as a parameter.

```
$~ sudo rmmod hello-world
```

Let's check out message again:

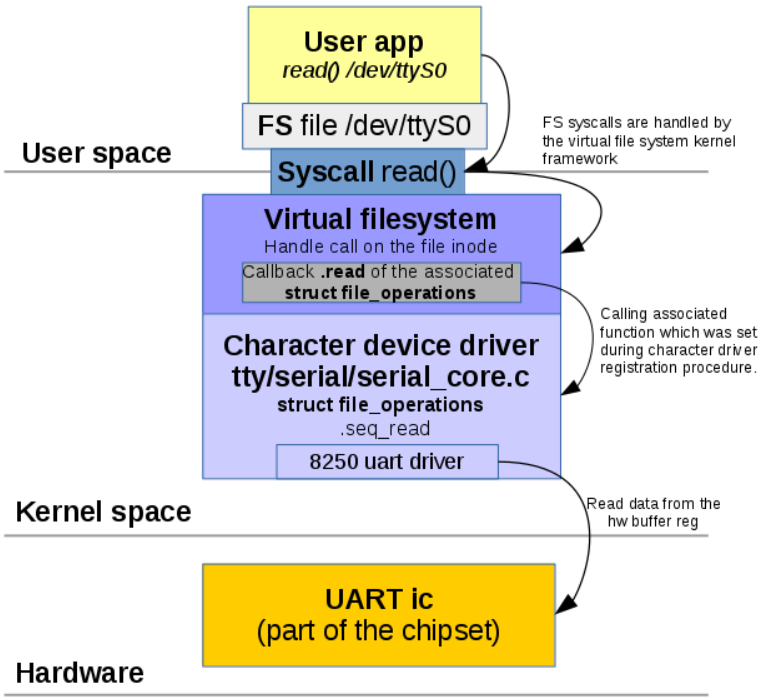
```
$~ sudo dmesg
...
[ 4867.502807] Hello, World!
[ 5003.403651] Goodbye, World!
```

V. A SIMPLE LINUX CHARACTER DEVICE DRIVER

1. Introduction to character device

A character device is one of the simplest ways to communicate with a module in the Linux kernel. These devices are presented as special files in a /dev directory and support direct reading and writing of any data, byte by byte, like a stream.

Actually, most of the pseudo-devices in /dev are character devices: serial ports, modems, sound, and video adapters, keyboards, and some custom I/O interfaces. Userspace programs can easily open, read, write, and custom control requests with such device files.



Let's take a look at the /dev folder. This directory contains the special device files for all the devices:

```
$~ ls -l /dev/ttyS0
total 0
crw-------  1 root root     10, 235 Nov 16 11:36 autofs
drwxr-xr-x  2 root root         280 Nov 16 11:36 block
crw-------  1 root root     10, 234 Nov 16 11:36 btrfs-control
drwxr-xr-x  3 root root          60 Nov 16 11:36 bus
drwxr-xr-x  2 root root        3360 Nov 16 11:36 char
crw-------  1 root root      5,    1 Nov 16 11:36 console
lrwxrwxrwx  1 root root          11 Nov 16 11:36 core -> /proc/kcore
...
```

Symbol C, in the beginning, means that this device is a character device. Also, we can find here two strange numbers, ex: 10 and 235 for the `autofs` driver. This is a Major and Minor number of this device.

Inside the Linux kernel, every device is identified not by a symbolic name but by a unique number – the device's major number. This number assigning by the kernel during device registration. Every device driver can support multiple "sub-devices". For example, a serial port adapter may contain two hardware ports. Both of these ports are handled by the same driver, and they share one Major number. But inside this driver, each of these ports is also identified by the unique number, and this is a device Minor number. The driver's code assigns minor numbers, and the developer of this driver may select any suitable values.

2. Character Device Operations

We'll look at the various operations a driver can perform on the devices it manages. The device is identified internally by a file structure, and the kernel uses the *file_operations* structure to access the driver's functions.

The *file_operations* structure is defined in *linux/fs.h,* and holds pointers to functions defined by the driver that perform various operations on the device. Each field of the structure corresponds to the address of some function defined by the driver to handle a requested operation:

```
struct file_operations {
        struct module *owner;
        loff_t (*llseek) (struct file *, loff_t, int);
        ssize_t (*read) (struct file *, char *, size_t, loff_t *);
        ssize_t (*write) (struct file *, const char *, size_t, loff_t *);
        int (*readdir) (struct file *, void *, filldir_t);
        unsigned int (*poll) (struct file *, struct poll_table_struct *);
        int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
        int (*mmap) (struct file *, struct vm_area_struct *);
        int (*open) (struct inode *, struct file *);
        int (*flush) (struct file *);
        int (*release) (struct inode *, struct file *);
        int (*fsync) (struct file *, struct dentry *, int datasync);
        int (*fasync) (int, struct file *, int);
        int (*lock) (struct file *, int, struct file_lock *);
         ssize_t (*readv) (struct file *, const struct iovec *, unsigned long,
            loff_t *);
         ssize_t (*writev) (struct file *, const struct iovec *, unsigned long,
            loff_t *);
    };
```

Some operations will not be implemented by a driver. For example, a driver that handles a video card won't need to read from a directory structure. The corresponding entries in the file_operations structure should be set to NULL. To keep our code clean, we can use the following declaration:

```
struct file_operations fops = {
   .read = device_read,
   .write = device_write,
   .open = device_open,
   .release = device_release
};
```

3. Registering A Device

As we already know, any char device is accessed through device files, usually located in /dev.. The major number tells you which driver handles which device file. The minor number is used only by the driver itself to differentiate which device it's operating on, just in case, the driver handles more than one device.

Adding a driver to your system means registering it with the kernel. This is synonymous with assigning it a major number during the module's initialization:

```
int alloc_chrdev_region(dev_t *dev, unsigned int firstminor, unsigned int count,
char *name);
```

where `dev` is output parameter for first assigned number, `firstminor` is first of the requested range of minor numbers (e.g., 0), `count` is a number of minor numbers required, and name – the associated device's name driver. The major number will be chosen dynamically and returned (along with the first minor number) in `dev`.

Next, it's time to initialize a new character device and set file_operations with `cdev_init`:

```
void cdev_init(struct cdev *cdev, const struct file_operations *fops);
```

where struct `cdev` represents a character device and is allocated by this function.

Now add the device to the system:

```
int cdev_add(struct cdev *p, dev_t dev, unsigned count);
```

Finally – create a device file node and register it with *sysfs*:

```
struct device * device_create(struct class *class, struct device *parent, dev_t
devt, const char *fmt, ...);
```

4. Device I/O functions

To interact with our device file, we need to set a few functions to the struct file_operations.

```
static int mychardev_open(struct inode *inode, struct file *file)
{
    printk("MYCHARDEV: Device open\n");
    return 0;
```

```
}

static int mychardev_release(struct inode *inode, struct file *file)
{
    printk("MYCHARDEV: Device close\n");
    return 0;
}

static long mychardev_ioctl(struct file *file, unsigned int cmd, unsigned long
arg)
{
    printk("MYCHARDEV: Device ioctl\n");
    return 0;
}

static ssize_t mychardev_read(struct file *file, char __user *buf, size_t count,
loff_t *offset)
{
    printk("MYCHARDEV: Device read\n");
    return 0;
}

static ssize_t mychardev_write(struct file *file, const char __user *buf, size_t
count, loff_t *offset)
{
    printk("MYCHARDEV: Device write\n");
    return 0;
}
```

4. Combine together

Let's create a complete driver program (main.c) and check whether it is working or not:

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/cdev.h>
#include <linux/device.h>
#include <linux/kernel.h>
#include <linux/uaccess.h>
#include <linux/fs.h>

#define MAX_DEV 2

static int mychardev_open(struct inode *inode, struct file *file);
static int mychardev_release(struct inode *inode, struct file *file);
static long mychardev_ioctl(struct file *file, unsigned int cmd, unsigned long
arg);
static ssize_t mychardev_read(struct file *file, char __user *buf, size_t count,
loff_t *offset);
static ssize_t mychardev_write(struct file *file, const char __user *buf, size_t
count, loff_t *offset);
```

```
static const struct file_operations mychardev_fops = {
    .owner      = THIS_MODULE,
    .open       = mychardev_open,
    .release    = mychardev_release,
    .unlocked_ioctl = mychardev_ioctl,
    .read       = mychardev_read,
    .write       = mychardev_write
};

struct mychar_device_data {
    struct cdev cdev;
};

static int dev_major = 0;
static struct class *mychardev_class = NULL;
static struct mychar_device_data mychardev_data[MAX_DEV];

static int mychardev_uevent(struct device *dev, struct kobj_uevent_env *env)
{
    add_uevent_var(env, "DEVMODE=%#o", 0666);
    return 0;
}

static int __init mychardev_init(void)
{
    int err, i;
    dev_t dev;

    err = alloc_chrdev_region(&dev, 0, MAX_DEV, "mychardev");

    dev_major = MAJOR(dev);

    mychardev_class = class_create(THIS_MODULE, "mychardev");
    mychardev_class->dev_uevent = mychardev_uevent;

    for (i = 0; i < MAX_DEV; i++) {
        cdev_init(&mychardev_data[i].cdev, &mychardev_fops);
        mychardev_data[i].cdev.owner = THIS_MODULE;

        cdev_add(&mychardev_data[i].cdev, MKDEV(dev_major, i), 1);

                device_create(mychardev_class,  NULL,  MKDEV(dev_major,  i),  NULL,
"mychardev-%d", i);
    }

    return 0;
}

static void __exit mychardev_exit(void)
{
    int i;
```

```
    for (i = 0; i < MAX_DEV; i++) {
        device_destroy(mychardev_class, MKDEV(dev_major, i));
    }

    class_unregister(mychardev_class);
    class_destroy(mychardev_class);

    unregister_chrdev_region(MKDEV(dev_major, 0), MINORMASK);
}

static int mychardev_open(struct inode *inode, struct file *file)
{
    printk("MYCHARDEV: Device open\n");
    return 0;
}

static int mychardev_release(struct inode *inode, struct file *file)
{
    printk("MYCHARDEV: Device close\n");
    return 0;
}

static long mychardev_ioctl(struct file *file, unsigned int cmd, unsigned long
arg)
{
    printk("MYCHARDEV: Device ioctl\n");
    return 0;
}

static ssize_t mychardev_read(struct file *file, char __user *buf, size_t count,
loff_t *offset)
{
    uint8_t *data = "Hello from the kernel world!\n";
    size_t datalen = strlen(data);

    printk("Reading device: %d\n", MINOR(file->f_path.dentry->d_inode->i_rdev));

    if (count > datalen) {
        count = datalen;
    }

    if (copy_to_user(buf, data, count)) {
        return -EFAULT;
    }

    return count;
}

static ssize_t mychardev_write(struct file *file, const char __user *buf, size_t
count, loff_t *offset)
{
    size_t maxdatalen = 30, ncopied;
    uint8_t databuf[maxdatalen];
```

```
    printk("Writing device: %d\n", MINOR(file->f_path.dentry->d_inode->i_rdev));

    if (count < maxdatalen) {
        maxdatalen = count;
    }

    ncopied = copy_from_user(databuf, buf, maxdatalen);

    if (ncopied == 0) {
        printk("Copied %zd bytes from the user\n", maxdatalen);
    } else {
        printk("Could't copy %zd bytes from the user\n", ncopied);
    }

    databuf[maxdatalen] = 0;

    printk("Data from the user: %s\n", databuf);

    return count;
}

MODULE_LICENSE("GPL");
MODULE_AUTHOR("BOSCH LINUX Dev Course");

module_init(mychardev_init);
module_exit(mychardev_exit);
```

And compile using the following Makefile:

```
BINARY     := mychardev
KERNEL      := /lib/modules/$(shell uname -r)/build
ARCH        := x86
C_FLAGS     := -Wall
KMOD_DIR    := $(shell pwd)
TARGET_PATH := /lib/modules/$(shell uname -r)/kernel/drivers/char

OBJECTS := main.o

ccflags-y += $(C_FLAGS)

obj-m += $(BINARY).o

$(BINARY)-y := $(OBJECTS)

$(BINARY).ko:
    make -C $(KERNEL) M=$(KMOD_DIR) modules

install:
    cp $(BINARY).ko $(TARGET_PATH)
    depmod -a
```

```
uninstall:
    rm $(TARGET_PATH)/$(BINARY).ko
    depmod -a

clean:
    make -C $(KERNEL) M=$(KMOD_DIR) clean
```

Then compile our driver and load into kernel:

```
$ make && sudo insmod mychardev.ko
```

Question: Find two new devices: /dev/mychardev-0 and /dev/mychardev-1,