# Programming Assignment one

**COMP3033 OS 1001-1004 2st semester 2021-2022**

Do the assignment at the end of this document (on page 11). This assignment will involve **designing two kernel modules in C**. Once you are finished, write a report, copy the code of the second module to the end of the report, Put your name and student ID number on your report. Also, add some pictures of your program running in the report. Write a paragraph to describe how you solve this problem. Did you encounter any problems? how do you fix it? In addition, upload the C files of both modules we ask you to design in this assignment on iSpace.

Late homework assignments will not be accepted unless you have a valid written excuse (medical, etc.). You must do this assignment alone. No teamwork or "talking with your friends" will be accepted. No copying from the Internet. Cheating means zero.

You can do this assignment on a computer running on Linux (Ubuntu or CentOS, or others on VM VirtualBox) only.

## Introduction to Linux Kernel Modules

In this project, you will learn how to create a kernel module and load it into the Linux kernel. You will then modify the kernel module so that it creates an entry in the /proc file system. The project can be completed using the Linux virtual machine that is available with this text. Although you may use any text editor to write these C programs, you will have to use the ***terminal*** application to compile the programs, and you will have to enter commands on the command line to manage the modules in the kernel.

As you'll discover, the advantage of developing kernel modules is that it is a relatively easy method of interacting with the kernel, thus allowing you to write programs that directly invoke kernel functions. It is important for you to keep in mind that you are indeed writing ***kernel code*** that directly interacts with the kernel. That normally means that any errors in the code could crash the system! However, since you will be using a virtual machine, any failures will at worst only require rebooting the system.

### I. Kernel Modules Overview

The first part of this project involves following a series of steps for creating and inserting a module into the Linux kernel.

You can list all kernel modules that are currently loaded by entering the command

```
lsmod
```

This command will list the current kernel modules in three columns: name, size, and where the module is being used.

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

/* This function is called when the module is loaded. */
int simple_init(void)
{
   printk(KERN_INFO "Loading Kernel Module\n");
   return 0;
}

/* This function is called when the module is removed. */
void simple_exit(void)
{
   printk(KERN_INFO "Removing Kernel Module\n");
}

/* Macros for registering module entry and exit points.
*/
module_init(simple_init);
module_exit(simple_exit);
MODULE LICENSE("GPL");
MODULE DESCRIPTION("Simple Module");
MODULE AUTHOR("SGG");
```

Kernel module simple.c.

The program in above Figure illustrates a very basic kernel module that prints appropriate messages when it is loaded and unloaded.

The function **simple_init()** is the **module entry point,** which represents the function that is invoked when the module is loaded into the kernel. Similarly, the **simple_exit()** function is the

**module exit point**—the function that is called when the module is removed from the kernel.

The module entry point function must return an integer value, with 0 representing success and any other value representing failure. The module exit point function returns void. Neither the module entry point nor the module exit point is passed any parameters. The two following macros are used for registering the module entry and exit points with the kernel:

```
module_init(simple_init)
module_exit(simple_exit)
```

Notice in the figure how the module entry and exit point functions make calls to the printk() function. printk() is the kernel equivalent of printf(), but its output is sent to a kernel log buffer whose contents can be read by the dmesg command. One difference between printf() and printk() is that printk() allows us to specify a priority flag, whose values are given in the <linux/printk.h> include file. In this instance, the priority is KERN INFO, which is defined as an ***informational*** message.

The final lines—MODULE_LICENSE(), MODULE_DESCRIPTION(), and MODULE_AUTHOR()—represent details regarding the software license, description of the module, and author. For our purposes, we do not require this information, but we include it because it is standard practice in developing kernel modules.

This kernel module simple.c is compiled using the Makefile accompanying the source code with this project. To compile the module, enter the following on the command line:

```
make
```

The compilation produces several files. The file simple.ko represents the compiled kernel module. The following step illustrates inserting this module into the Linux kernel.

## II. Loading and Removing Kernel Modules

Kernel modules are loaded using the insmod command, which is run as follows:

```
sudo insmod simple.ko
```

To check whether the module has loaded, enter the lsmod command and search for the module simple. Recall that the module entry point is invoked when the module is inserted into the kernel. To check the contents of this message in the kernel log buffer, enter the command

```
dmesg
```

You should see the message "Loading Module."

Removing the kernel module involves invoking the rmmod command
(notice that the .ko suffix is unnecessary):

```
sudo rmmod simple
```

Be sure to check with the dmesg command to ensure the module has been removed.

Because the kernel log buffer can fill up quickly, it often makes sense to clear the buffer periodically. This can be accomplished as follows:

```
sudo dmesg -c
```

Proceed through the steps described above to create the kernel module and to load and unload the module. Be sure to check the contents of the kernel log buffer using dmesg to ensure that you have followed the steps properly.

As kernel modules are running within the kernel, it is possible to obtain values and call functions that are available only in the kernel and not to regular user applications. For example, the Linux include file <linux/hash.h> defines several hashing functions for use within the kernel. This file also defines the constant value GOLDEN_RATIO_PRIME (which is defined as an unsigned long).
This value can be printed out as follows:

```
printk(KERN_INFO "%lu\n", GOLDEN_RATIO_PRIME);
```

As another example, the include file <linux/gcd.h> defines the following function

```
unsigned long gcd(unsigned long a, unsigned b);
```

which returns the greatest common divisor of the parameters a and b.

Once you are able to correctly load and unload your module, complete the following additional steps:

1. Print out the value of GOLDEN_RATIO_PRIME in the simple init() function.
2. Print out the greatest common divisor of 3,300 and 24 in the simple_exit() function.

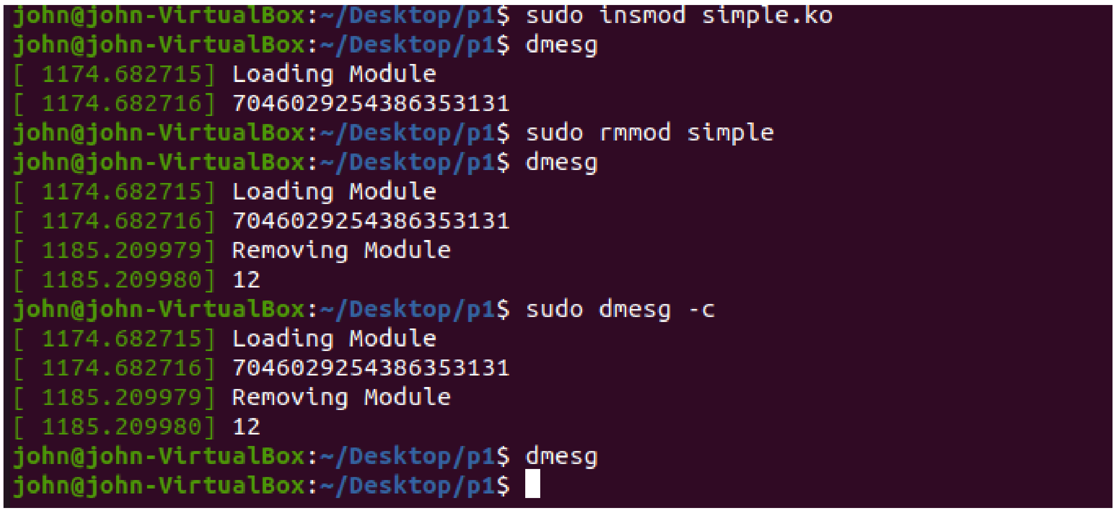
The code is modified as follows:

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

/* This function is called when the module is loaded. */
int simple_init(void)
{
   printk(KERN_INFO "Loading Kernel Module\n");
   printk(KERN_INFO "%lu\n", GOLDEN_RATIO_PRIME);
   return 0;
}

/* This function is called when the module is removed. */
void simple_exit(void)
{
   printk(KERN_INFO "Removing Kernel Module\n");
   printk(KERN_INFO "%ld\n", gcd(3300, 24));
}

/* Macros for registering module entry and exit points.
*/
module_init(simple_init);
module_exit(simple_exit);
MODULE LICENSE("GPL");
MODULE DESCRIPTION("Simple Module");
MODULE AUTHOR("SGG");
```



```
john@john-VirtualBox:~/Desktop/p1$ sudo insmod simple.ko
john@john-VirtualBox:~/Desktop/p1$ dmesg
[ 1174.682715] Loading Module
[ 1174.682716] 7046029254386353131
john@john-VirtualBox:~/Desktop/p1$ sudo rmmod simple
john@john-VirtualBox:~/Desktop/p1$ dmesg
[ 1174.682715] Loading Module
[ 1174.682716] 7046029254386353131
[ 1185.209979] Removing Module
[ 1185.209980] 12
john@john-VirtualBox:~/Desktop/p1$ sudo dmesg -c
[ 1174.682715] Loading Module
[ 1174.682716] 7046029254386353131
[ 1185.209979] Removing Module
[ 1185.209980] 12
john@john-VirtualBox:~/Desktop/p1$ dmesg
john@john-VirtualBox:~/Desktop/p1$ 
```

As compiler errors are not often helpful when performing kernel development, it is important to compile your program often by running make regularly. Be sure to load and remove the kernel module and check the kernel log buffer using dmesg to ensure that your changes to simple.c are working properly.

In Section 1.4.3, we described the role of the timer as well as the timer interrupt handler. In Linux, the rate at which the timer ticks (the **tick rate**) is the value HZ defined in <asm/param.h>. The value of HZ determines the frequency of the timer interrupt, and its value varies by machine type and architecture.

For example, if the value of HZ is 100, a timer interrupt occurs 100 times per second, or every 10 milliseconds. Additionally, the kernel keeps track of the global variable jiffies, which maintains the number of timer interrupts that have occurred since the system was booted. The jiffies variable is declared in the file <linux/jiffies.h>.

1. Print out the values of jiffies and HZ in the simple_init() function.
2. Print out the value of jiffies in the simple_exit() function.

Here is the solution code:

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

```
#include <linux/module.h>
#include <linux/hash.h>
#include <linux/gcd.h>
#include <asm/param.h>
#include <linux/jiffies.h>

/* This function is called when the module is loaded. */
int simple_init(void)
{
   printk(KERN_INFO "Loading Kernel Module\n");
   printk(KERN_INFO "%lu\n", GOLDEN_RATIO_PRIME);
   printk(KERN_INFO "%d, %ld\n", HZ, jiffies);
   return 0;
}

/* This function is called when the module is removed. */
void simple_exit(void)
{
   printk(KERN_INFO "Removing Kernel Module\n");
   printk(KERN_INFO "%ld\n", gcd(3300, 24));
   printk(KERN_INFO "%ld\n", jiffies);

}

/* Macros for registering module entry and exit points.
*/
module_init(simple_init);
module_exit(simple_exit);
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Simple Module");
MODULE_AUTHOR("SGG");
```

---

Before proceeding to the next set of exercises, consider how you can use the different values of jiffies in simple_init() and simple_exit() to determine the number of seconds that have elapsed since the time the kernel module was loaded and then removed.

One possible solution could be as follows:

---

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

```
#include <linux/hash.h>
#include <linux/gcd.h>
#include <asm/param.h>
#include <linux/jiffies.h>

long int old_jiffies;

/* This function is called when the module is loaded. */
int simple_init(void)
{
   printk(KERN_INFO "Loading Kernel Module\n");
   printk(KERN_INFO "%lu\n", GOLDEN_RATIO_PRIME);
   printk(KERN_INFO "%d, %ld\n", HZ, jiffies);
   old_jiffies = jiffies;
   return 0;
}

/* This function is called when the module is removed. */
void simple_exit(void)
{
   printk(KERN_INFO "Removing Kernel Module\n");
   printk(KERN_INFO "%ld\n", gcd(3300, 24));
   printk(KERN_INFO "%ld\n", jiffies);
   printk(KERN_INFO "%ld\n", (jiffies-old_jiffies)/HZ);



}

/* Macros for registering module entry and exit points.
*/
module_init(simple_init);
module_exit(simple_exit);
MODULE LICENSE("GPL");
MODULE DESCRIPTION("Simple Module");
MODULE AUTHOR("SGG");
```

---

## III. The /proc File System

The /proc file system is a "pseudo" file system that exists only in kernel memory and is used primarily for querying various kernel and per-process statistics.

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

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/proc_fs.h>
#include <asm/uaccess.h>

#define BUFFER_SIZE 128
#define PROC_NAME "hello"

//for kernel version (5.6.0) or above
//find your Linux system kernel version:
//$ sudo uname -a   or  $ cat /proc/version
#define HAVE_PROC_OPS

ssize_t proc_read(struct file *file, char  *usr_buf,
size_t count, loff_t *pos);

#ifdef HAVE_PROC_OPS
static struct proc_ops ops = {
    .proc_read = proc_read,
};
#else
static struct file_operations ops = {
  .owner = THIS_MODULE,
  .read = proc_read,
};
#endif

/* This function is called when the module is loaded. */
int proc_init(void)
{
  /* creates the /proc/hello entry */
  proc_create(PROC_NAME, 0666, NULL, &ops);
  return 0;
}
/* This function is called when the module is removed. */
void proc_exit(void)
{
  /* removes the /proc/hello entry */
  remove_proc_entry(PROC_NAME, NULL);
}
```

This exercise involves designing kernel modules that create additional entries in the /proc file system involving both kernel statistics and information related to specific processes.

We begin by describing how to create a new entry in the /proc file system.

The following program example (named hello.c) creates a /proc entry named /proc/hello. If a user enters the command

**cat /proc/hello**

The infamous Hello World message is returned.

```
/* This function is called each time /proc/hello is read */
ssize_t proc_read(struct file *file, char _user *usr_buf,
size_t count, loff_t *pos)
{
  int rv = 0;
  char buffer[BUFFER_SIZE];
  static int completed = 0;
  if (completed) {
    completed = 0;
    return 0;
  }
  completed = 1;
  rv = sprintf(buffer, "Hello World\n");
  /* copies kernel space buffer to user space usr buf */
  raw_copy_to_user(usr_buf, buffer, rv);
  return rv;
}
module_init(proc_init);
module_exit(proc_exit);
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Hello Module");
MODULE_AUTHOR("SGG");
```

In the module entry point proc_init(),we create the new /proc/hello entry using the proc_create() function. This function is passed proc_ops, which contains a reference to a struct file_operations. This struct initializes the .owner and .read members. The value of .read is the name of the function proc_read() that is to be called whenever /proc/hello is read.

Examining this proc_read() function, we see that the string "Hello World\n" is written to the variable buffer where buffer exists in kernel memory. Since /proc/hello can be accessed from user space, we must copy the contents of buffer to user space using the kernel function raw_copy_to_user(). This function copies the contents of kernel memory buffer to the variable usr_buf, which exists in user space.

Each time the /proc/hello file is read, the proc_read() function is called repeatedly until it returns 0, so there must be logic to ensure that this function returns 0 once it has collected the data (in this case, the string "Hello World\n") that is to go into the corresponding /proc/hello file.

Finally, notice that the /proc/hello file is removed in the module exit point proc_exit() using the function remove_proc_entry().

## IV. Assignment

This assignment will involve designing two kernel modules:

1. Design a kernel module that creates a /proc file named /proc/jiffies that reports the current value of jiffies when the /proc/jiffies file is read, such as with the command

```
cat /proc/jiffies
```

Be sure to remove /proc/jiffies when the module is removed.

2. Design a kernel module that creates a proc file named /proc/seconds that reports the number of elapsed seconds since the kernel module was loaded. This will involve using the value of jiffies as well as the HZ rate. When a user enters the command

```
cat /proc/seconds
```

Your kernel module will report the number of seconds that have elapsed since the kernel module was first loaded. Be sure to remove /proc/seconds when the module is removed.

Good Luck!