Operating System

project 3

Department: 土木5B
ID: 0511330
Name: 劉紘華
Vedio Link: https://youtu.be/-Z5-M9gFLBI

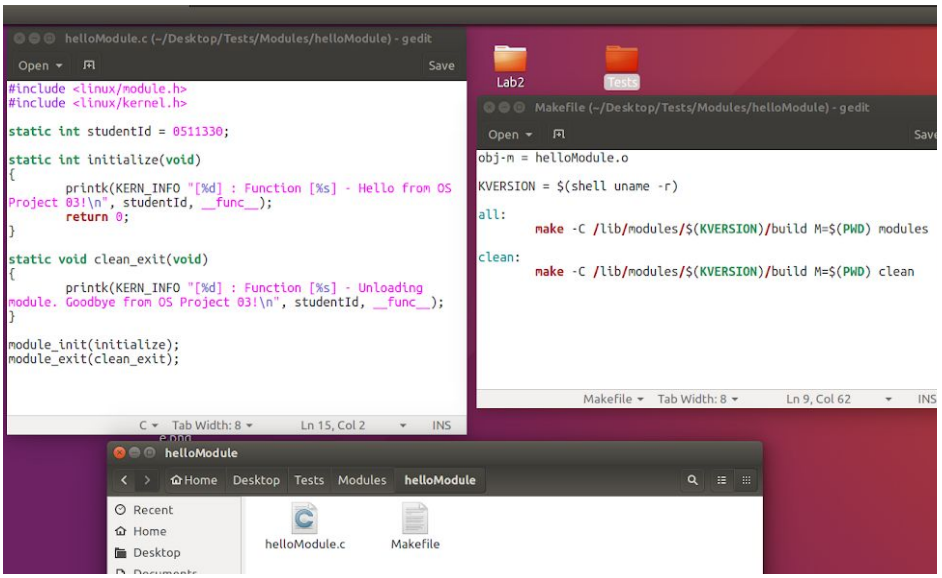
Ojective:

To help students to get familiar with Linux Kernel Modules. Students will learn the definition, usage and how to implement custom Linux Kernel Modules, and how to mount and unmount them.

[Section 1-2] Load the module manually

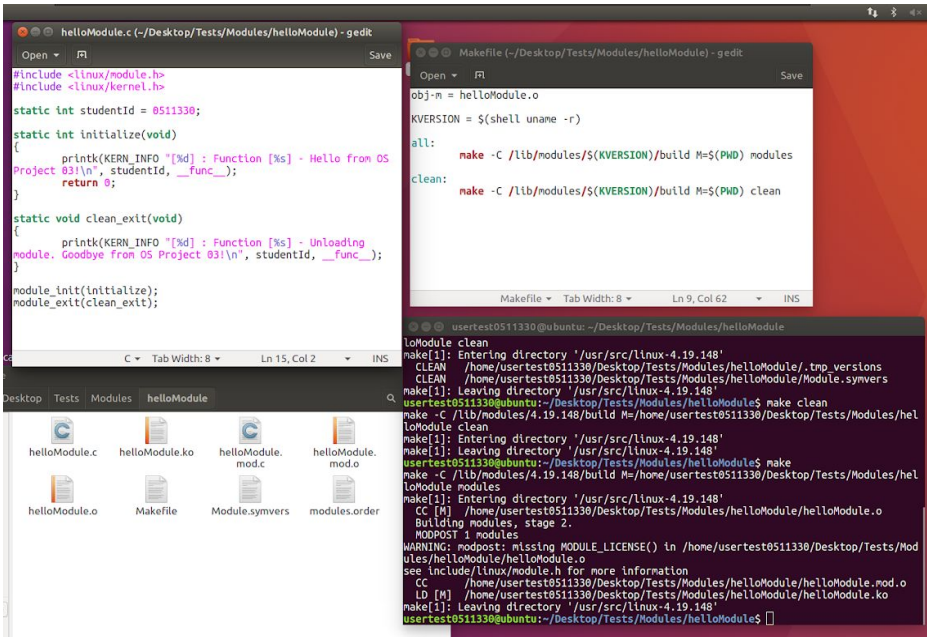
[Screenshot 1] helloModule code

The make file is used to tell the compiler the module needed to compile.



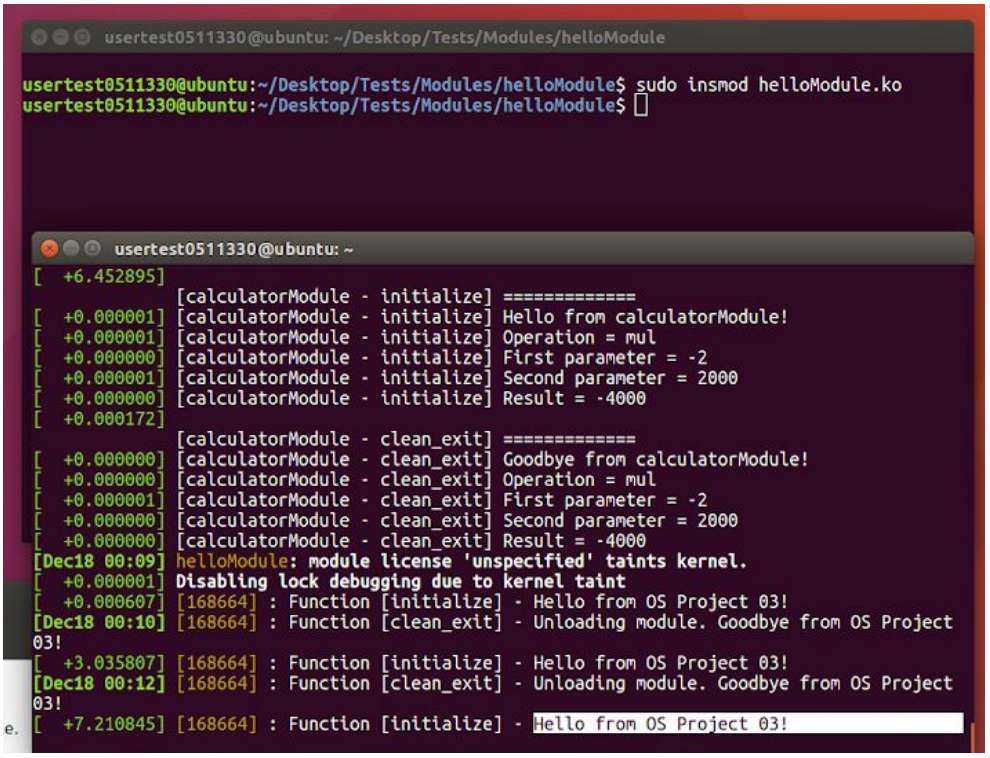
[Screenshot 2] make clean & make

Build the module by the command **make** under the module directory. We can clean the object code of the module through the command **make clean**.



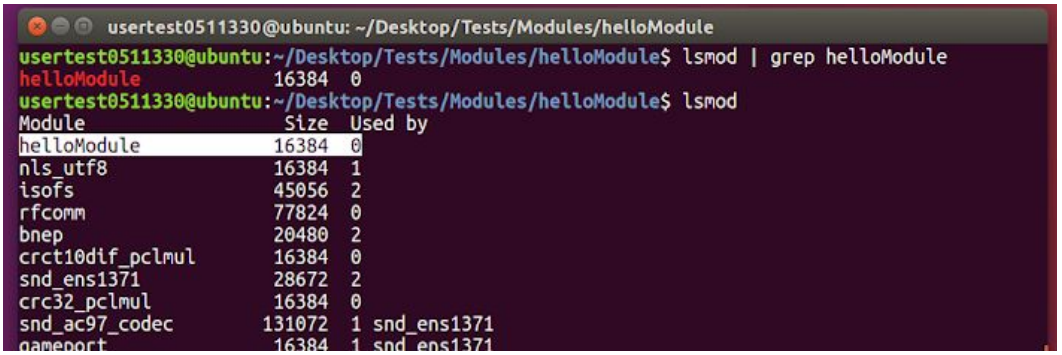
[Screenshot 3]

Use the command **dmesg -wH** to see the messages printed by our module in the **kernel ring buffer**.



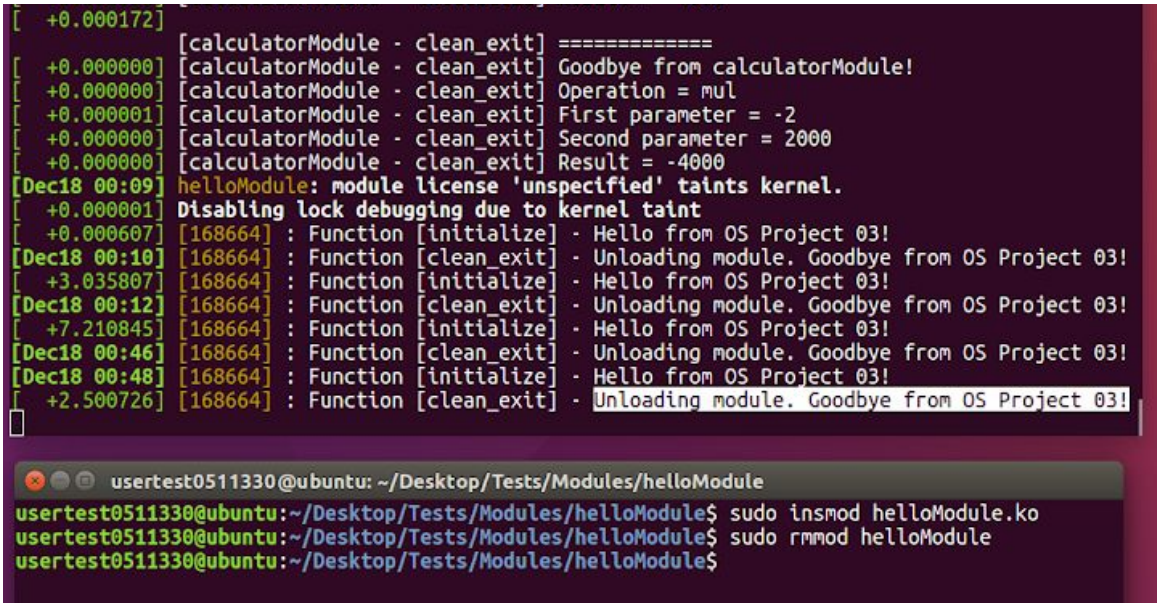
[Screenshot 4]

To see all modules are loading currently , use the command **lsmod**.
To check whether the module is loaded successfully, use the command **lsmod | grep helloModule**



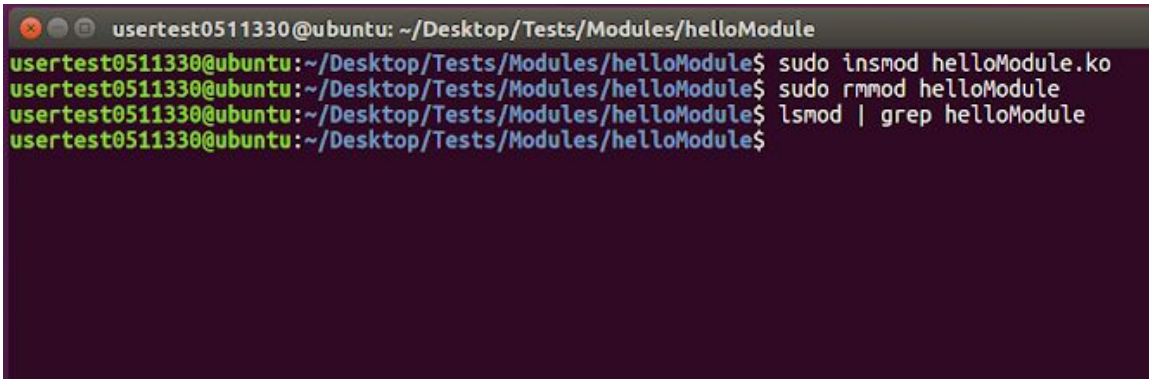
[Screenshot 5]

Unload the module by the command **rmmod hellodModule**, the messages printed by module shown in the **ring buffer**



[Screenshot 6]

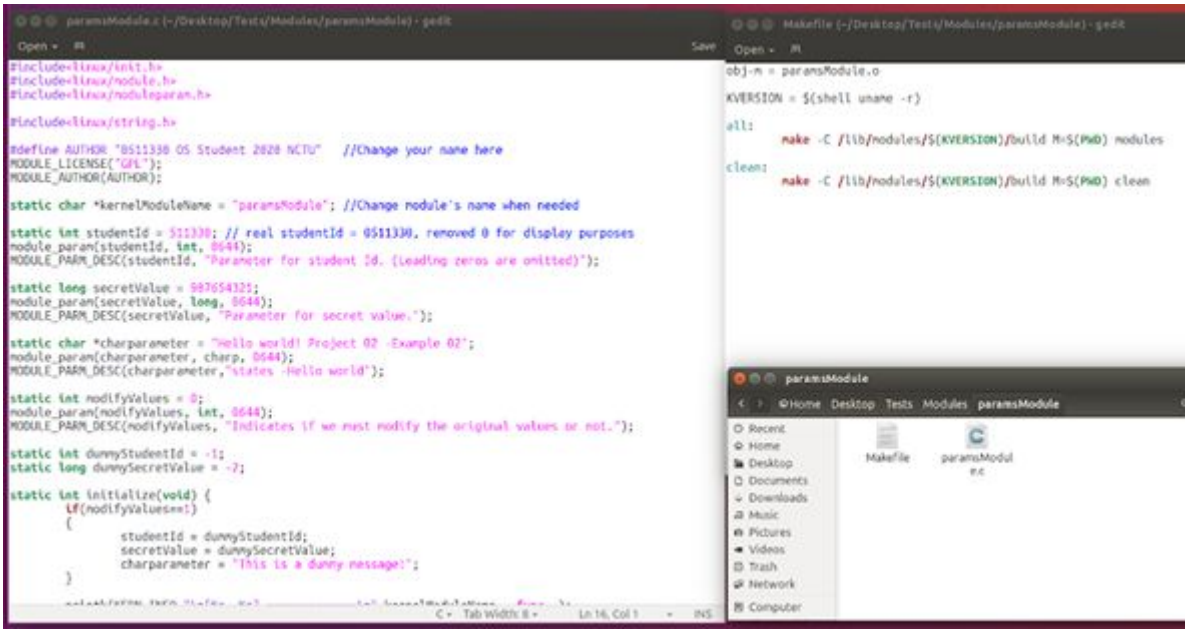
When the module has been unloaded, **lsmod | grep helloModule** returns nothing, which implies our module unloads successfully.



[Section 1-2]

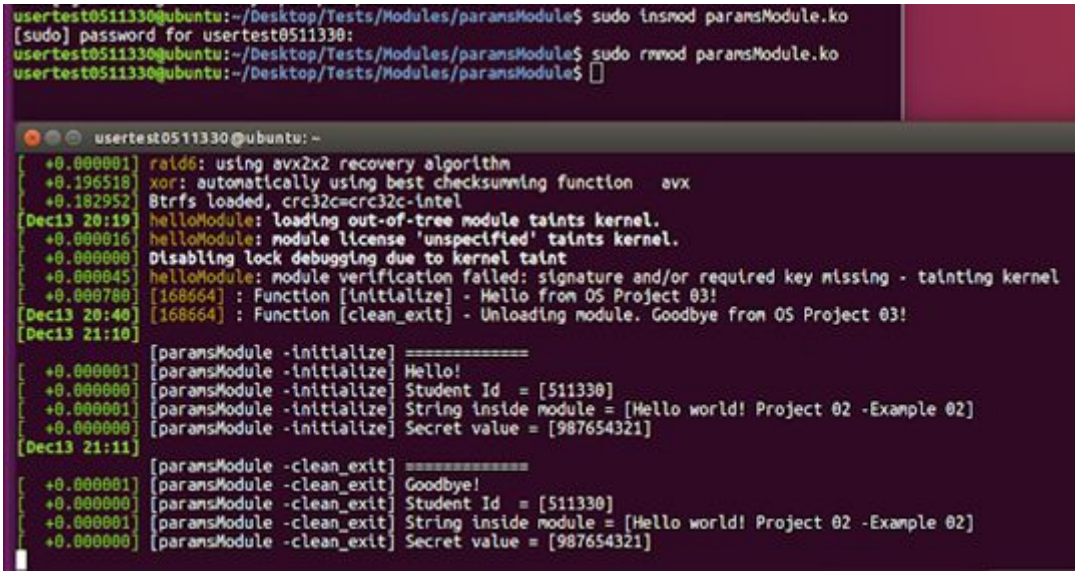
[Screenshot 7]

The paramModule contains parameters, the module will react to different moves depending on different values of parameters.



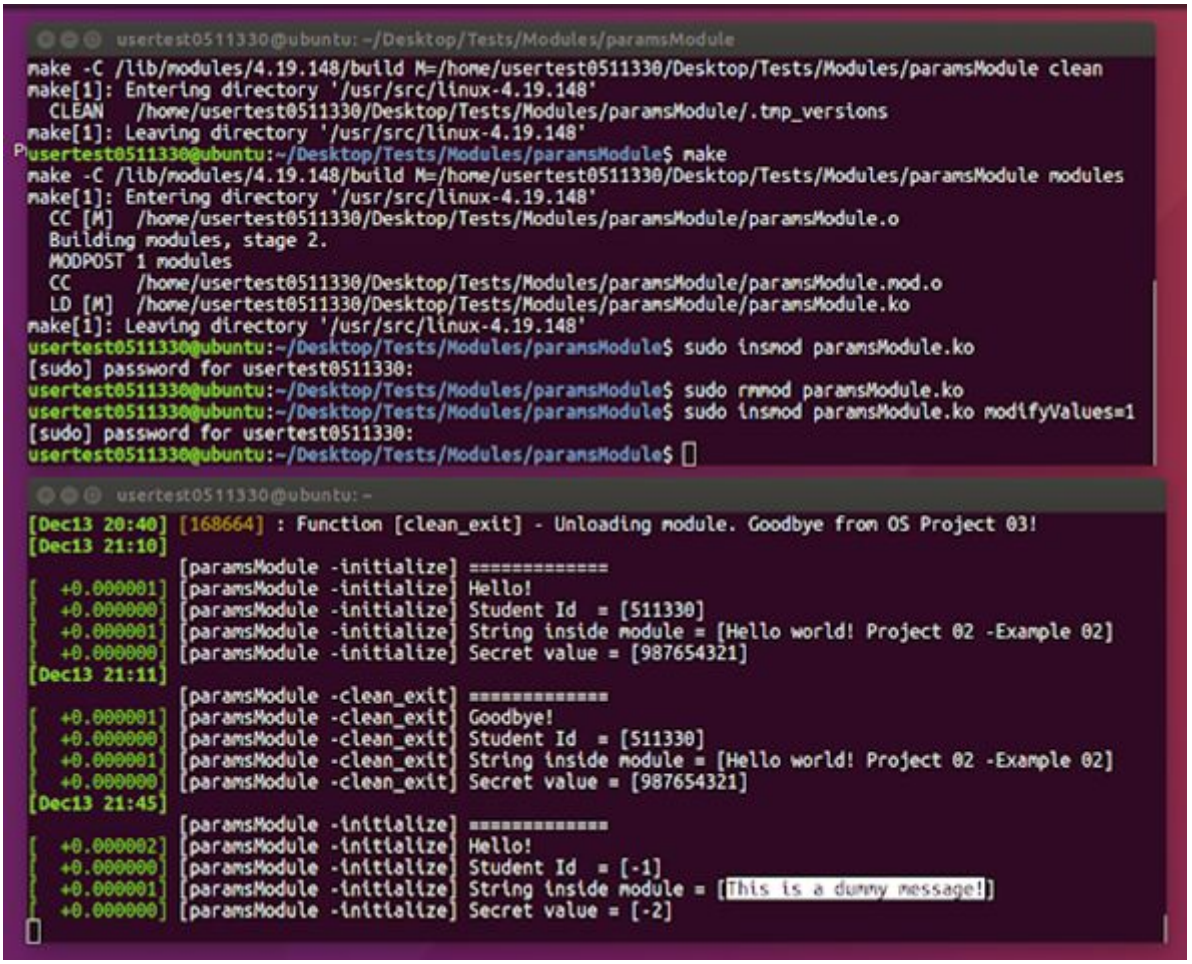
[Screenshot 8]

If we are **not sending the parameters**, the default value will remain the same all the time when loading and unloading the module.



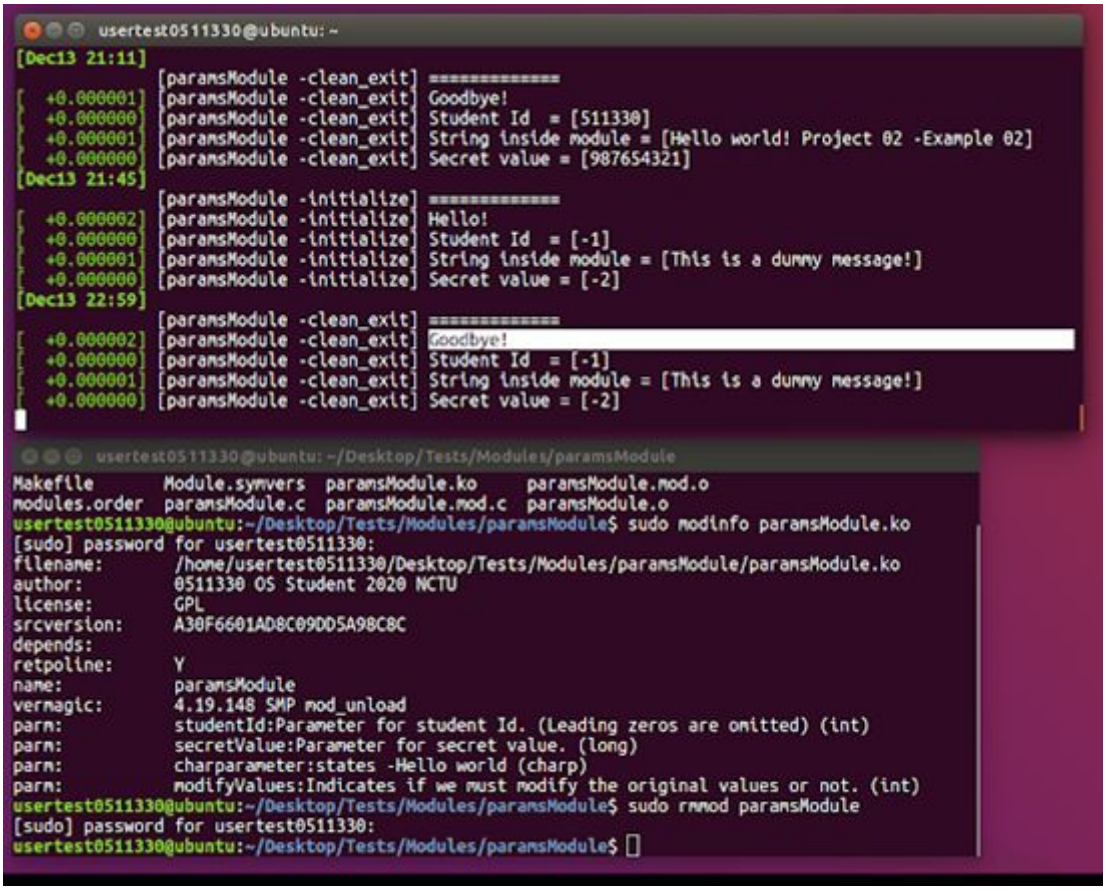
[Screenshot 9]

If we load the module with parameters (**modifyValues=1**), the module will receive it and react differently (e.g. **studentId** will be changed).



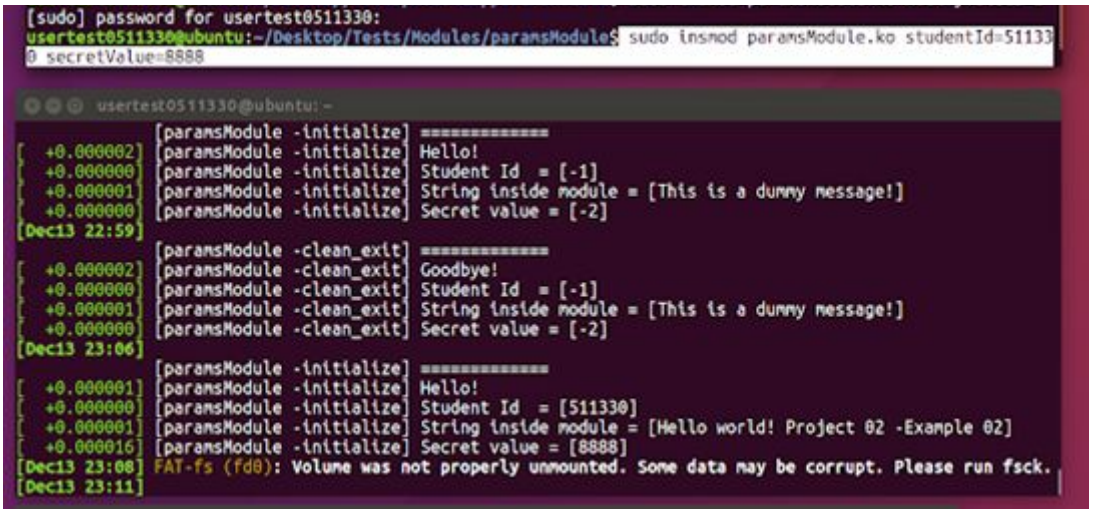
[Screenshot 10]

With parameters triggering, we changed other parameters permanently. So the changed value will also be updated in the text file under the **/paramsModule/parameter** folder.
Use **modinfo paramsModule.ko** to show more information about the module.



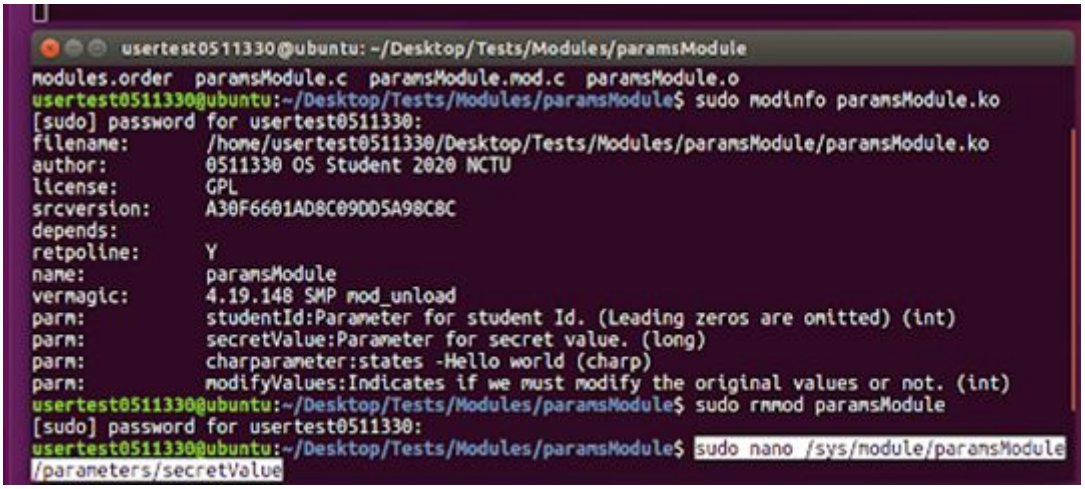
[Screenshot 11]

We load the module again with more parameters:**studentId = 511330**, **secretValue = 8888.**



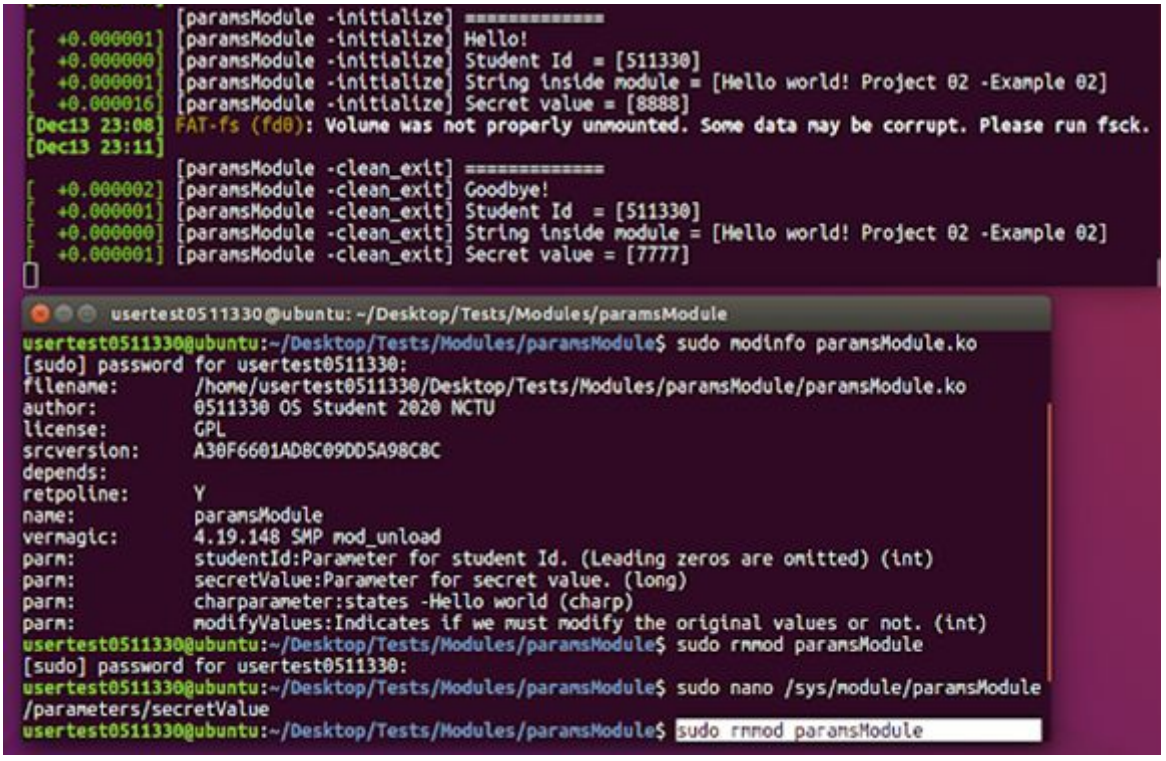
[Screenshot 12] modify scretValue to 7777

Further, we modify the parameter **secretValue to 7777**, which is saved in the text file. By command:
nano /sys/module/paramsModule/parameters/secreteValue.



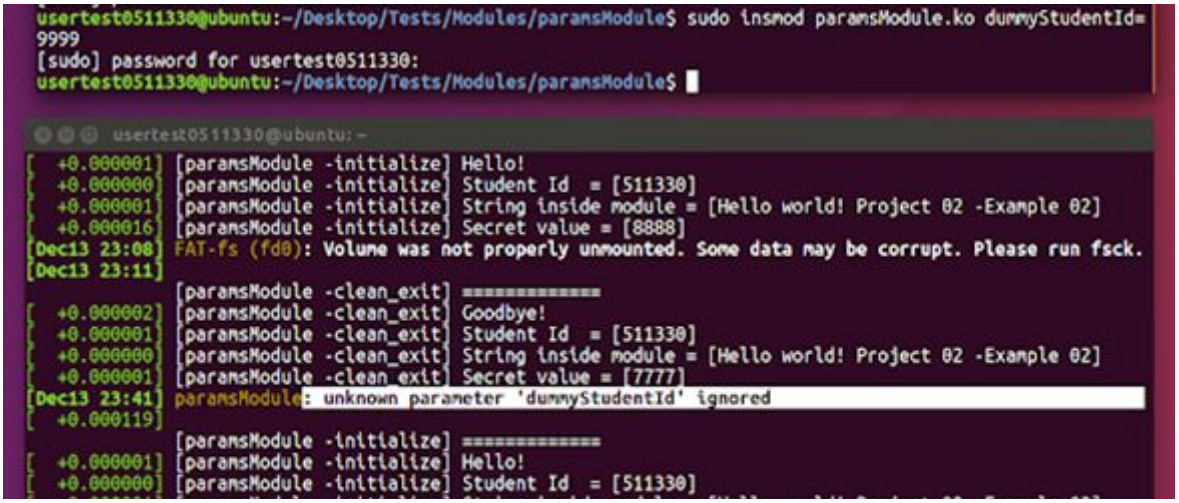
[Screenshot 13] remove module, scretValue = 7777

When we unload the module, the message shows that secreValue has been changed to 7777, which is shown in the ring buffer.

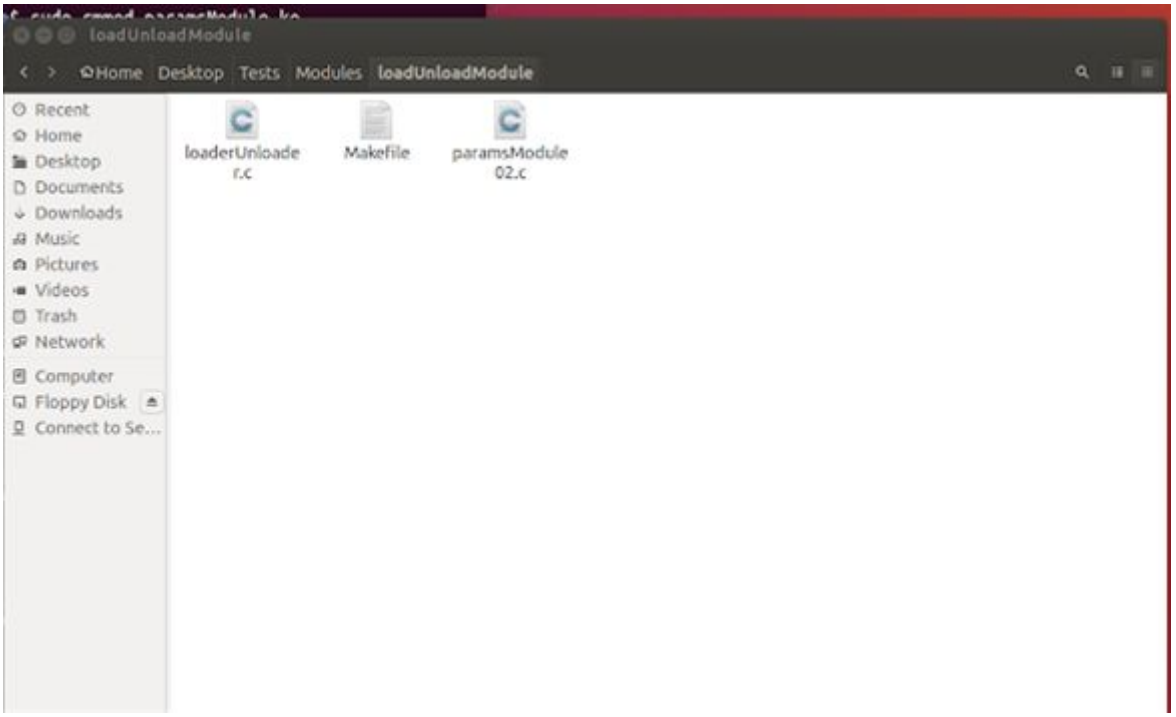


[Screenshot 14] modinfo showing “Unknown parameter” messages

The module will only receive the parameters, when we try to send a value to a variable which is not a parameter of the module, it shows an ignored message.



[Screenshot 15]



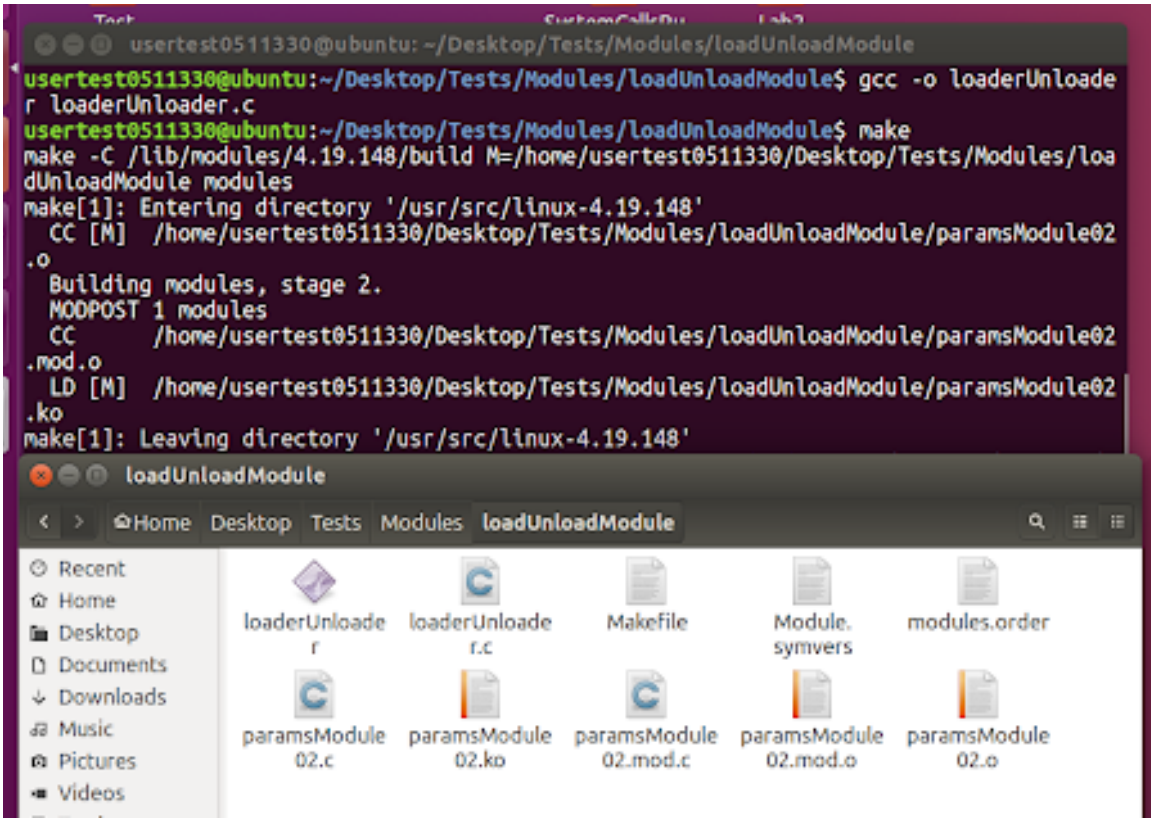
[Screenshot 16]

We use the **loaderUnloader** program to load the **paramsModule02** dynamically, with parameter **“studentId=511330”**.
We also define the system calls which load and unload the module as **init_module, delete_modue** by macro, to improve code readability.

```
nloader.c (~/Desktop/Tests/Modules/loadUnloadModule) - gedit
Open
/ Author: Heyward Liu
/ StudentID: 0511330

nt main(int argc, char**argv) {

        printf("\nThis is a dynamic loader and unloader for a kernel module!\n");

        // Module information
        const char *moduleName = "paramsModule02.ko";
        const char *moduleNameNoExtension = "paramsModule02";
        const char *paramsNew = "studentId=511330"; // Use your StudentID without leading 0

        int fd, use_finit;
        size_t image_size;
        struct statst;
        void *image;

        //Section -Module loading -BEGIN
        fd = open(moduleName, O_RDONLY);

        printf("Loading module [%s] with parameters [%s]...\n",moduleNameNoExtension,paramsNew);
        fstat(fd, &st);
        image_size = st.st_size
        image = malloc(image_size);
        read(fd, image, image_size);
        if(init_module(image, image_size, paramsNew) != 0) {
                perror("init_module");
                return EXIT_FAILURE;
        }
        printf("Module is mounted!\n");
        //Section -Module loading -END


        //  At this point the module is mounted.
        //  You can check it with $ lsmod | grep <name of module without extension>
        //  You can access its variables in /sys/module/<name of module without textension>/parameters
        //  WARNING: IF YOU MODIFY THE VARIABLES WITHOUT FOLLOWING THE CORRECT
        //      DATATYPEYOUR MODULE WILL GET LOCKED AND YOU MUSST FIX THE VARIABLES
        //      AND RESTART YOUR MACHINE

        printf("\n[Press ENTER to continue]\n");
        getchar();

        //Section -Module unloading -BEGIN
        printf("Unmounting module...\n");

        if(delete_module(moduleNameNoExtension, O_NONBLOCK) != 0) {
                perror("delete_module");
                return EXIT_FAILURE;
        }

        close(fd);
        printf("Module is unmounted!\n");
        printf("Cleaning...\n");

        free(image);
        //Section -Module unloading -END

        printf("Done!\n");

        return 0;
}
```

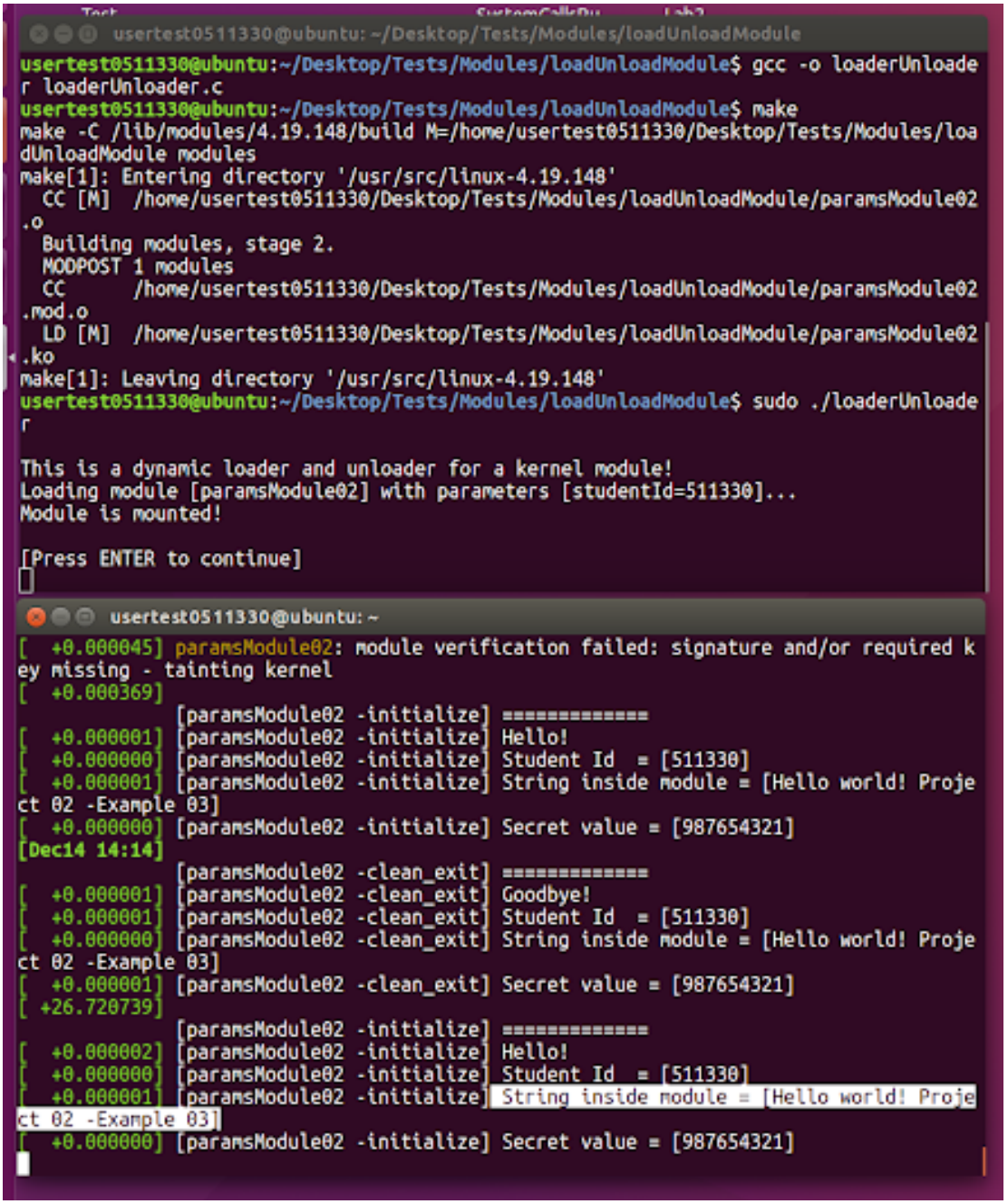
[Screenshot 17]

Using gcc to compile the **loaderUnloader** C file to the object code, and build it into kernel object code by command **$make**.



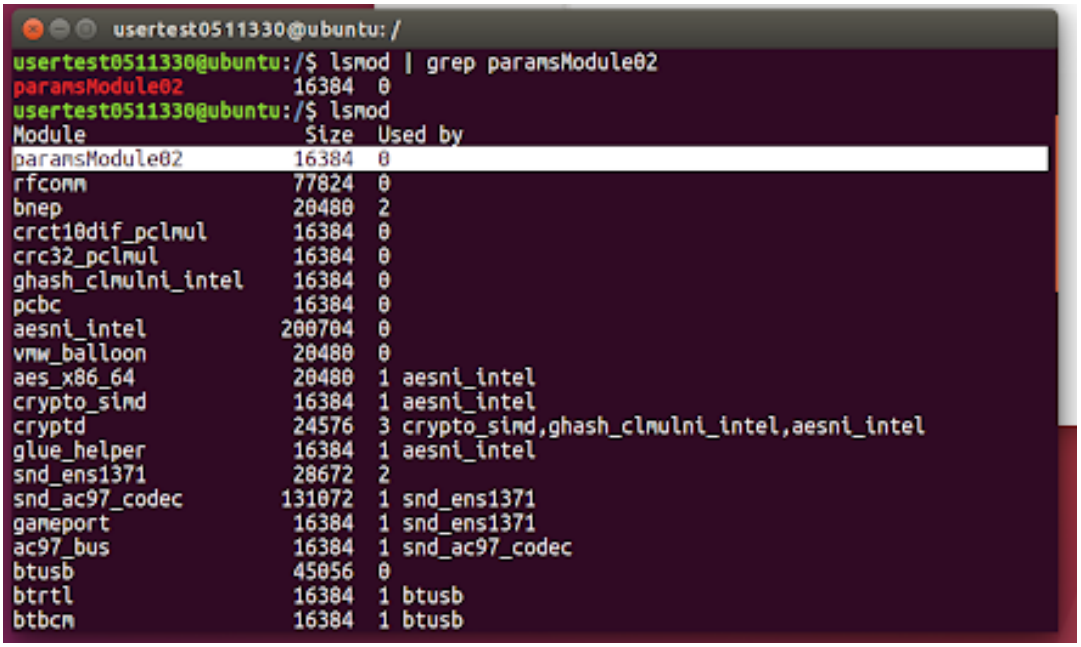
[Screenshot 18]

By running the **loaderUnloader** program, the module **paramsModule02** has been loaded with a received parameter: **studentId = 511330**.
Now the loaderUnloader program is stuck in statement - **getchar()**.
We can check the messages shown in **ring buffer** by the command **$dmesg -wH**



[Screenshot 19]

Further, we can also check it is loaded successfully by the commands **lsmod** / **lsmod | grep paramsMoudle02.**



[Screenshot 20]

[upper terminal]

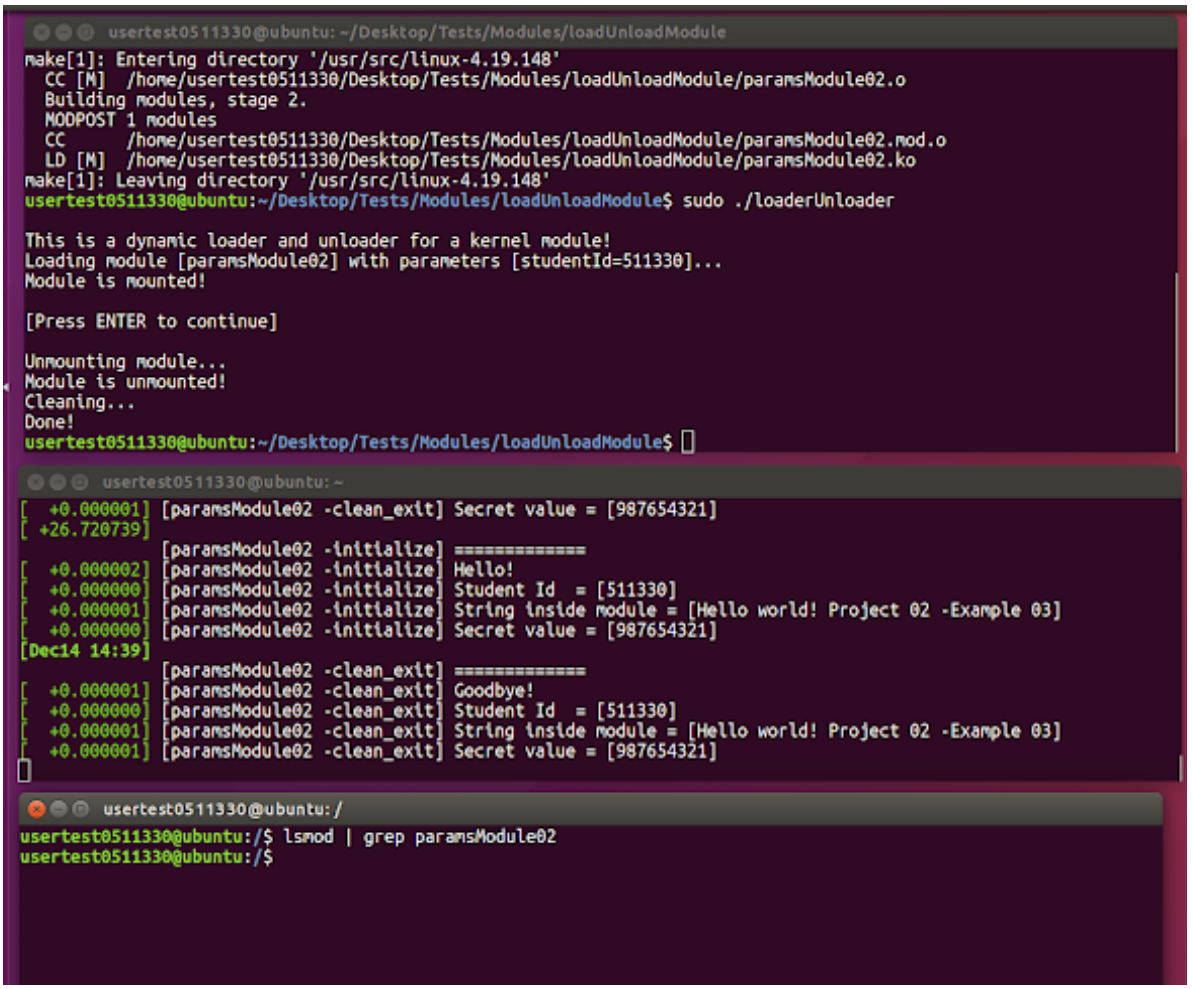
When we proceed to continue the loadUnload file which is currently stuck in **getchar()**, our module will be unloaded.

[middle terminal]

Messages in the **ring buffer** shows our module was cleaned up.

[lower terminal]

Further, we check whether it was unloaded successfully by the command **lsmod | grep paramsModule02**, and we get nothing in return. It means our module is no longer in the module list.



[Screenshot 21] Implemented by loading the module

We figure out the result by loading **calculatorModule** in our calculator program, and send operand and operation to the module to get the result. All operations can be divided it into 3 major moves:

1.LoadModule
2.ReadResult
3.UnLoadModule

```
long addition (char* AllParams)
{
    long result = 0;
    LoadModule(AllParams);
    result = ReadResult();
    UnLoadModule();

    return result;
}

long substraction (char* AllParams)
{
    long result = 0;
    LoadModule(AllParams);
    result = ReadResult();
    UnLoadModule();

    return result;
}

long multiplication (char* AllParams)
{
    long result = 0;
    LoadModule(AllParams);
    result = ReadResult();
    UnLoadModule();

    return result;
}
```

[Screenshot 22] Parameters sended to the module

All parameters sended to the module are concatenated to a string called **AllParams**, which includes operation, input1 and input2.

```
//concat All parameters into a string

sprintf(paramsNew, "operationParam=%s firstParam=%d secondParam=%d",
operation, input1, input2);
```

[Screenshot 22] Load & UnLoad

To our convenience and improving code readability, We define the function by macro. So we can load the **calculatorModule** by calling **init_module** and unload it by **delete_module.**

```
#define init_module(module_image, len, param_values)
syscall(__NR_init_module, module_image, len, param_values)

#define finit_module(fd, param_values, flags)
syscall(__NR_finit_module, fd, paramvalues, flags)

#define delete_module(name, flags) syscall(__NR_delete_module,
name, flags)
```

Further, we pack all procedures when loading / unloading a module into functions called **LoadModule / UnLoadModule** in order to reuse the code.

```
void LoadModule(char* Params) {
    int fd, use_finit;
    size_t image_size;
    struct stat st;
    void *image;

    fd = open(moduleName, O_RDONLY);
    fstat(fd, &st);
    image_size = st.st_size;
    image = malloc(image_size);
    read(fd, image, image_size);
    if(init_module(image, image_size, Params) != 0)
{
        perror("init_module");
    }
}
```

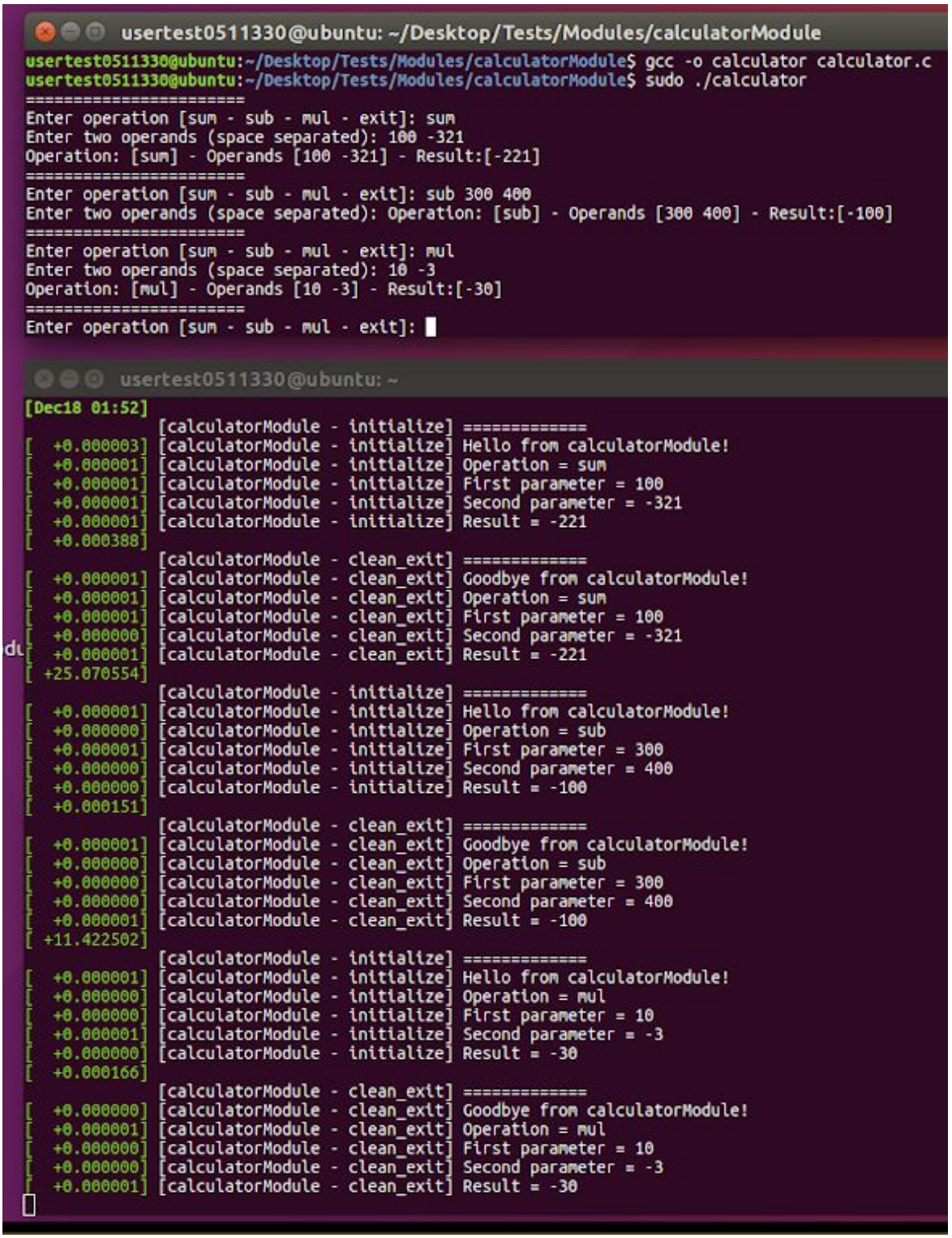
```
void UnLoadModule() {
    if(delete_module(moduleNameNoExtension,
O_NONBLOCK)!=0) {
        perror("delete_module");
    }
}
```

[Screenshot 23] Read the Result

When the result is figured out by the module, it is saved into a text file called **resultParam** under **/sys/module/calculatorModule/parameters.** We simply use **fopen** to open the file and **fscanf** to read the result into our variable.

```
long ReadResult() {
    long result=0;
    FILE *file = fopen("/sys/module/calculatorModule/parameters/resultParam", "r");
    if(file) {
        fscanf(file, "%ld", &result);
    }
    return result;
}
```

[Screenshot 24] Final result



[Quetions]

1. What is a static / dynamic kernel module ?
   What is the other name of a dynamic kernel module?
   What are the differences between system calls and dynamic kernel modules?

   (i) **static modules** are compiled as part of the base kernel and it is available at any time.
   dynamic modules are compiled as modules separately and loaded based on user demand.

   (ii) **Loadable Kernel Modules**

   (iii) (a) **System calls** will be in the kernel all the time as the API to serve all user programs; **Loadable Kernel Modules** will load only when a specific needs.
   (b) **System calls** need to be compiled as part of kernel, **LKM** can be compiled at any time.
   (c) We will load **LKM** into the kernel by a **system call**.

2. Why does adding a system call require kernel recompilation, while adding a kernel module does not?

   System call is the part of the kernel as APIs to provide OS services to user programs. Fundamentally, system call is treated as kernel.
   Kernel module is more like a user program. It's only loaded into the kernel when we need it. It makes the kernel more flexible.

3. What are the commands insmod, rmmod and modinfo for?

   insmod dummyModule.ko means insert the dummyModule.
   rmmod dummyModule.ko means remove the dummyModule.
   modinfo dummyModule.ko shows the information of dummyModule.

4. Write the usage of the following commands:

    a. **module_init** is a **macro** that defines which function is to be called at insertion time.

    b. **module_exit** is a **macro** that defines which funcion to be called at removal time.

    c. **MODULE_LICENSE** is the license of the module, a **macro** which allows LKM to declare their license to the world.

    d. **module_param** is a **macro that** indicates the variables are module parameters and its data type.

    e. **MODULE_PARAM_DESC** is a **macro** that is used to document arguments that the module can take, and will be written into **modinfo.**

5. What do the following commands mean

    a. **cat <filename>** is used to print the whole file on the terminal.

    b. **ls -l** print the file-access date of each file of the whole directory.

    c. **dmesg** shows kernel log, and **-wH** means that **keep reading** a new log in **human readable form**.

    d. **lsmod** will list all modules that have been loaded currently.

    e. **lsmod | grep <module name>** return the specific module state which has been loaded currently, and return nothing if the module has not been loaded.

6. What does 0644 mean in module_param()?

    It refers to permission of accessing the parameters file, 0644 means modules allow us to access the parameter files freely.

7. What happens if the initialization function of the module returns -1? What type error do you get?

   When initialization returns -1, it means that loading the module fails. We will receive **Operation not permitted** from the terminal. It's because insert the module requiring the root permission.

8. Two of the variables don't show in the **modinfo** in section 1.3**, why?**

   Because only parameters will show on the **modinfo**, two of them are not parameters.

9. What is the **/sys/module** folder for?

   To place code, parameters and other stuff of modules which have been loaded into the kernel currently.

10. What is charp in section 1.3?

    You will use a type of **charp** to define a module parameter that takes a string.