# Practice Lab 2 - Linux Kernel Module and Process Representation in Linux
## Operating Systems SFWRENG 3SH3, Winter 2023
## Prof. Neerja Mhaskar

**Lab Format:** Practice labs will be posted a day before or on the day of the lab on the course website. You can choose to solve it beforehand and come in with your solutions and check the correctness of your solution with your TA.

During your lab hours, the TAs will also be available to answer any questions you may have on your assignments.

### Outline

In this lab you will learn the following:
1. How to create a kernel module and loading/removing them from the Linux kernel.
2. How processes are represented in Linux and how to access members of the idle/swapper process.

This lab should be completed using the Linux virtual machine that you installed in Practice Lab 1. Although you may use any editor to write these C programs, you will have to use the *terminal* application (on the virtual machine) 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.

### Part 1 - Linux Kernel Modules

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.

The program `simple.c` posted on the course website under content->practice labs 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 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 posted on the course website under content->practice labs. 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.

**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.

**Part – 2: Process Representation in Linux**

**You can use the file (simple.c) you created under Part-1 of this lab to complete Part-2.**

Processes in Linux are represented by the `task_struct` data structure as shown in Chapter 3 lecture slides. This data structure is defined in the sched.h header file. Take a look at the definition of this data structure and study the following members of this structure: `pid, state, flag, rt_priority` (runtime priority), `policy` (process policy) and `tgid` (Task group id).

In this part of the lab, you will write a Linux kernel module which outputs the values of the above listed members to the kernel log buffer for the `init` task (also called the swapper/idle process) in a Linux system, when the module is loaded into the kernel.

Print the output of the `dmesg` command to a file and show it to your TA.

**Note:**

1. The `task_struct` data structure for the Linux swapper (idle process) is named `init_task` and it has a `pid = 0.` It is a task scheduled to run when no other process exists to run on the system.
2. If you are using the `simple.c` file, you need to only write a procedure `print_init_PCB()` that prints the required members of init_task. Then call this function in `simple_init().`

**SAMPLE OUTPUT**

```
------------------------------------------------------------------------------------------
Loading Module
[ 2300.008729] init_task pid:0
[ 2300.008733] init_task state:0
[ 2300.008736] init_task flags:2097152
[ 2300.008740] init_task runtime priority:0
[ 2300.008743] init_task process policy:0
[ 2300.008745] init_t
```

**Extra Material to do on your own.**

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.

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 of the textbook the role of the timer as well as the timer interrupt handler is described. 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.