

Assignment 3: Introduction to Linux Kernel Modules (Tutorial) (v0.13)

Note: Use of Ubuntu 22.04 is required for this assignment.

The purpose of this assignment is to introduce Linux Kernel programming in the context of writing a Linux Kernel module. Kernel code executes in kernel (privileged mode) where program instructions can command the CPU to perform any operation allowed by the CPU's architecture. This allows any instruction to be executed, any I/O operation to be initiated, and any area of memory accessed directly.

The host operating system is the base operating system of the computer which is started on boot time. The guest operating system is the operating system of any virtual machine that runs on the host. It is best to complete this tutorial on a virtual machine. In case of problems, it is easier to restart a virtual machine, than to restart the host system.

Assignment Submission

This assignment is accompanied by an online Canvas quiz which captures the answers for the questions in this assignment. After completing the activities described below, to complete the assignment and receive a grade, log into Canvas, and complete the Assignment #3 Quiz.

For this assignment, the assumption is that the following packages have already been installed. If they are not installed, they can be installed with:

```
sudo apt update
sudo apt install make binutils gcc
sudo apt install --reinstall gcc-12
```

1. Download kernel module starter code.

To begin, download the tar.gz archive containing a basic Hello World Linux kernel module.

First, create a directory for this assignment, and then download the tar gz:

Kernel modules will FAIL TO COMPILE if source code is placed under paths with spaces. In general spaces in filenames is a Windows/MAC convention and is uncommon on Linux. Please do not use spaces in directory or filenames.

```
mkdir a3
cd a3
wget http://faculty.washington.edu/wlloyd/courses/tcss422/assignments/hello\_module.tar.gz
tar xzf hello_module.tar.gz
cd hello_module
ls -l
```

The hello_module contains "helloModule.c", which includes source code for the most basic Linux kernel

module. Inspect the code. The module consists of an initialization method and a cleanup method. The initialization method is triggered automatically as an “event” when the module is loaded. The cleanup method is triggered when the kernel module is unloaded. Kernel modules are loaded dynamically into the operating system. They are not run as user programs on the command line. Consequently, they don’t have traditional I/O (e.g. stdin, stdout).

Note: Required #include files have already been included in hello_module.c for this assignment.

2. Compile, install, and test a Linux Kernel Module

Try compiling this module, and inspecting the resulting files:

```
make
```

You may receive the compiler warning as below:

```
make -C /lib/modules/6.5.0-28-generic/build M=/home/maliha/a3/hello_module modules
make[1]: Entering directory '/usr/src/linux-headers-6.5.0-28-generic'
warning: the compiler differs from the one used to build the kernel
The kernel was built by: x86_64-linux-gnu-gcc-12 (Ubuntu 12.3.0-1ubuntu1~22.04) 12.3.0
You are using:          gcc-12 (Ubuntu 12.3.0-1ubuntu1~22.04) 12.3.0
CC [M] /home/maliha/a3/hello_module/helloModule.o
MODPOST /home/maliha/a3/hello_module/Module.symvers
CC [M] /home/maliha/a3/hello_module/helloModule.mod.o
LD [M] /home/maliha/a3/hello_module/helloModule.ko
BTF [M] /home/maliha/a3/hello_module/helloModule.ko
Skipping BTF generation for /home/maliha/a3/hello_module/helloModule.ko due to unavailability of vmlinux
make[1]: Leaving directory '/usr/src/linux-headers-6.5.0-28-generic'
```

The compiler warning can be ignored as long as the kernel module compiles. Check if the module compiled:

```
ls -alt
```

If gcc-12 is NOT installed, the kernel module may fail to compile.

Check if gcc-12 is installed:

```
$ whereis gcc-12
gcc-12: /usr/bin/gcc-12 /usr/share/man/man1/gcc-12.1.gz
```

If gcc-12 is not installed, install it as described on page 1.

The newly compiled kernel module should appear at the top of the directory output.

Compilation produces a kernel object (ko) file. Load this file:

```
sudo insmod helloModule.ko
```

The “insmod” command must be run as superuser. This “inserts” a module into the running kernel. When we load the module the init() method is automatically called. Any “printk” statements are sent to the kernel log file. To view what has been written to the kernel log file, check the last 10 lines of /var/log//syslog using:

```
tail -n 10 /var/log/syslog
```

The module can be unloaded using the “rmmod” command. It is not necessary to include the “ko” file extension:

```
sudo rmmod helloModule
```

Check the kernel log after unloading the module:

```
tail -n 10 /var/log/syslog
```

Separate log messages are written by the init() and cleanup() methods.

For this tutorial, we will iteratively make changes to the helloModule kernel module. For each iteration we will “rmmod” the previous version, “insmod” the new version”, and check printk output to /var/log/syslog using the tail command.

Messages regarding kernel module verification can be ignored as this is a user provided test module.

3. Count the number of running Linux tasks

Only privileged kernel code can read kernel data structures.

Let’s access the task_struct data structure to inspect running processes and threads on the computer. Let’s add code to the init() method that is run when the module is loaded.

The task_struct data structure requires the <linux/sched/signal.h> include statement which defines an important helper function called `for_each_process()`. This include statement is ***already*** in the starter code.

```
#include<linux/sched/signal.h>
```

Now add source code for the function:

```
int proc_count(void)
{
    int i=0;
    struct task_struct *thechild;
    for_each_process(thechild)
        i++;
    return i;
}
```

Next, call this function by adding a printk() statement to init():

```
printk(KERN_INFO "There are %d running processes.\n", proc_count());
```

In this assignment, we leverage two Linux kernel data structures:

```
struct task_struct
defined in /usr/src/linux-headers-$(uname -r)/include/linux/sched.h
starting at line 630 (value for 5.8 Ubuntu 20.04 kernel, exact location may vary)
```

```
struct mm_struct
defined in /usr/src/linux-headers-$(uname -r)/include/linux/mm_types.h
starting at line 386 (value for 5.8 Ubuntu 20.04 kernel, exact location may vary)
```

Before recompiling, first check to ensure that these files are on your system with “ls”:

```
ls -l /usr/src/linux-headers-$(uname -r)/include/linux/sched.h
ls -l /usr/src/linux-headers-$(uname -r)/include/linux/mm_types.h
```

If you do not have these files, check if they are installed with:

```
sudo dpkg -l | grep linux-headers-$(uname -r)
```

If the result is blank, then try installing the required headers:

```
sudo apt install linux-headers-$(uname -r) linux-headers-$(uname -r)-generic
```

Then test if they are available:

```
sudo dpkg -l | grep linux-headers-$(uname -r)
```

If there are still problems obtaining the linux headers, please [contact the instructor](#).

Recompile and reload the kernel module and check output to the syslog.

<Q1> - How many processes do you see?

4. Inspecting Linux Task Codesize

Next, implement the function below to scan Linux processes and determine the code size for each of them.

Important Note about COPY-AND-PASTE:

If copying and pasting code from a PDF file, please note that some characters may not be copied correctly. It may be necessary to correct quote marks, subtract operators, etc. **The DOCX file is better for copy/paste.**

```
unsigned long code_memory(void)
{
    unsigned long totalmem=0;
    unsigned long thismem;
    struct task_struct *thechild;
    for_each_process (thechild)
    {
        printk(KERN_INFO "pid %d", thechild->pid);
        thismem=thechild->mm->end_code - thechild->mm->start_code;
        printk(KERN_INFO " has codesize %lu.\n", thismem);
        totalmem=totalmem+thismem;
    }
    return totalmem;
}
```

Now, call this function by adding a `printk()` statement to `init()`:

```
printk(KERN_INFO "Running processes have %lu bytes of code.\n", code_memory());
```

Recompile and reload the kernel module and check output to the `syslog`.

<Q2> – What happens when this kernel module is loaded?

Check the kernel log file.

```
tail -n 60 /var/log/syslog
```

Scroll up to view the full content of the log.

Look for an error message that says “BUG: unable to handle kernel”...

Examine the description of the error carefully to learn what has caused the issue.

>> Now, reboot and restore the Ubuntu virtual machine before continuing. <<

To proceed the bug needs to be corrected.

Not every process can be indexed with the `code_memory()` function.

Processes must be tested before running the code.

Line 9 of the function dereferences **two pointers**:

```
    thismem=thechild->mm->end_code - thechild->mm->start_code;
```

When the first pointer is not valid, it is not possible to dereference the second pointer.

When the second pointer is not valid, it is not possible to read any members of the `mm` (memory map).

The variable “`thechild`” is a `task_struct` pointer.

Let’s inspect what member “`mm`” is by looking at `sched.h` in the `task_struct` type definition. See the table above on page 3 for the location of the **sched.h include file**. Skip to approximately **line 858** in the file.

Line 858 assumes Ubuntu 22.04 with Linux kernel 5.19.0-42.

If using a different version of Ubuntu, this line number could vary.

You’ll be looking for the location where `struct task_struct` is defined.

It will look like:

```
struct task_struct {
#ifdef CONFIG_THREAD_INFO_IN_TASK
    /*
     * For reasons of header soup (see current_thread_info()), this
     * must be the first element of task_struct.
     */
    struct thread_info thread_info;
#endif
    volatile long state;    /* -1 unrunnable, 0 runnable, >0 stopped */
    void *stack;
```

To correct the bug, we must first check that `mm` is valid before using it.

<Q3> – What test should be made before dereferencing a member that is pointed to in C ?

Correct the issue in Q3, and then continue.

The correction should allow the function to run where it will print the codesize of every process to the /var/log/syslog log file.

If you experience trouble corrected the error, contact the instructor for a hint.

Next, for simplicity **combine** the two printk() statements in the function.

Remove the first printk() statement, and replace the second printk() with the following:

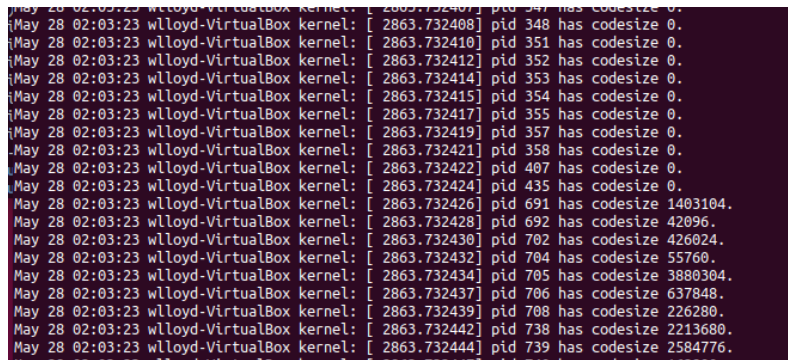
```
printk(KERN_INFO "pid %d has codesize %lu.\n", thechild->pid, thismem);
```

Recompile and reload the kernel module.

Test the output by inspecting the tail of the log file:

```
tail -n 250 /var/log/syslog
```

It should be possible to see processes with and without code:



<Q4> – What is the codesize of process #1? This is the Linux kernel’s init process, the parent of all processes.

Write down the codesize of process #1.

Now, in the Linux include files, search for the “mm” member of this data structure.

In the task_struct definition, the mm member may be defined around **line 647** of the file.

<Q5> – What is the data type for the mm member of task_struct data structure?

Provide name exactly as it appears in the header file. (case sensitive)

5. Inspecting Linux Task Datasize and Heapsize

Next, look up the structure for task_struct’s “mm” member.

This data type (struct) is defined in the **mm_types.h include file**.

Check the table above on page 3 for the location of this file.

This struct, defined in mm_types, spans many lines of code.

Look for the two members (fields) in the struct that we’ve used in the code above:

```
field: start_code  
field: end_code
```

Now copy the `code_memory()` function and create two new functions.
Name one function: "`data_memory()`" and the other "`heap_memory()`".

Replace the use of `start_code` and `end_code` in these functions.

In `data_memory`, use:

```
field: start_data  
field: end_data
```

In `heap_memory`, use:

```
field: start_brk  
field: brk
```

In the `init()` function modify the `printk` call to call all three memory functions:

```
printk(KERN_INFO "Running processes have %lu bytes of code, %lu bytes of data, and  
%lu bytes of heap.\n", code_memory(), data_memory(), heap_memory());
```

Rebuild and reinstall the kernel module, and then check the output to `/var/log/syslog`.

There will be a lot of lines. Try adjusting the number of lines returned by the `tail` command to data from all processes, including process #1.

For example:

```
$tail -n 750 /var/log/syslog
```

<Q6> - For PID #1, which segment type utilizes the smallest amount of memory space?

- A. Heap
- B. Data
- C. Code
- D. Stack
- E. None of the above

<Q7> - For all processes combined, which segment type utilizes the largest amount of memory space?

- A. Heap
- B. Data
- C. Code
- D. Stack
- E. None of the above

<Q8> - For all processes combined, which segment type utilizes the smallest amount of memory space?

- A. Heap
- B. Data
- C. Code
- D. Stack
- E. None of the above

<Q9> - Please copy and paste the complete output from your final Linux kernel module from /var/log/syslog into the box below for Question 9.

<Q10> - (optional) If experiencing difficulty copying-and-pasting the kernel module's output to the textbox for Question 9, please create a text file that captures the kernel module's output, and upload the file here.

To complete this assignment, to obtain credit, log in to Canvas, and complete the **Assignment 3 Quiz**.

Version	Date	Change
0.1	5/23/2024	Original Version
0.11	6/7/2024	Note added about gcc-12 and compiler warning
0.12	6/7/2024	Removed Microsoft "Smart Quotes"
0.13	6/7/2024	Removed Q4 ambiguity.

Please help improve this tutorial by reporting any errors or discrepancies found.