Jesse Roberts
Tony Rodriguez
Eli Hoehne

We have neither given nor received unauthorized assistance on this work.

The name of your virtual machine (VM): group_12_vm
The password of your virtual machine (VM): macrohard

Describe where the files can be found:
Navigate to the directory: /home/trodriguez/OS_P2
FILE TREE:

```
[root@localhost OS_P2]# tree
.
├── part1
│   ├── hello.c
│   └── Makefile
├── part2
│   ├── Makefile
│   └── print_self.c
├── part3
│   ├── Makefile
│   └── print_other.c
└── part4
    ├── code.c
    └── Makefile
```

Describe each file and the purpose it serves:
File Descriptions:
Above is a tree showing the file paths
part1 contains both the hello.c and the makefile for the associated file. The purpose of this is to write a hello world Linux kernel module.
part2 contains both the print_self.c and the makefile for the associated file. The purpose of this is to identify the current process at the user-level and print out various information about the process as well as all parent processes to init.
part3 contains both the print_other.c and the makefile for the associated file. The purpose of this is create a module that takes a process PID as its argument and outputs the above information (same as part2) of this process all the way back to init
part4 contains both the code.c and the makefile for the associated file. The purpose of this is to answer Part 4 #2 and see if the code from 20 years ago is still valid and viable to this day.

Provide any special instructions to access or run your program:
List of commands to run all parts:

In order to execute our program for part 1 you would follow these steps:

1. Use the Makefile as normal: make
2. Load our module like so: insmod hello.ko
3. Use the dmesg command as usual

4. To unload our module use: rmmod hello
5. Then run “make clean” command

In order to execute our program for part 2 you would follow these steps:

1. Use the Makefile as normal: make
2. Load our module like so: insmod print_self.ko
3. Use the dmesg command as usual
4. To unload our module use: rmmod print_self
5. Then run “make clean” command

In order to execute our program for part 3 you would follow these steps:

1. Use the Makefile as normal: make
2. Load our module like so: insmod print_other.ko target_pid=$(cat random.txt)
3. Use the dmesg command as usual

If you would like to run the module again and want it to use a different arbitrary PID then after removing the module and running make clean, manually remove the random_pid.txt file and follow steps 1 - 3 again.

A brief description about how you solved the problems and what you learned:

Over the course of the project, we faced various problems. One such problem was the "No such file or directory," error where the kernel headers required for building kernel modules couldnt be found. We ended up solving this issue by running yum update which update our OS to the newest version and fixed the issue. Another major issue we encountered was when trying to load our module for task 3. When we would run insmod with the kernel object and the parameter needed for task 3, our VM would freeze and time out. We found that the while loop that traverses the parent tree ended up becoming an infinite loop and we were able to solve this by rewriting the condition for the loop. Overall, this project allowed us to learn 2 major concepts. Firstly, we learned how to write a Linux kernel module and secondly, we were able to learn how to write makefiles for the kernel modules.

More thorough descriptions and problems for Part 1, Part 2, Part 3 and Part 4 are written below in the description above each part.

Part 1:

The first part of this assignment consisted of making a simple kernel module followed by scouring the provided makefile for errors. We decided to fix the makefile first in order to try to make the provided kernel module C file. Using the sample code provided in the assignment. We first needed to install the linux headers by using the yum install -y kernel-devel kernel-headers command inside the VM. This command allowed us to implement kernel modules correctly. The next step was to create a hello world kernel module. We took the provided code and made the hello.c file, we then changed the name of the methods from init_module(void) and cleanup_module(void) to static int hello_init(void) and static void hello_exit(void). This is because we were getting naming convention errors and redefinition of init_module and cleanup_module. These errors are

because these functions are already defined/named by the kernel headers. We found renaming these methods to be a suitable fix and eliminated the errors.

1. Do you see any errors when you make? If so, how to fix the error (5pts)?
   Yes, we received errors with the original makefile. When initially trying to use the makefile provided, we found a few errors which caused the makefile to fail. These errors were mainly missing separator errors (i.e. basic formatting issues). For example, the spacing around the += needed to be removed and as well as the all and clean commands needed to be tabbed not spaced. With these changes to the makefile we were able to compile the hello.c code using the makefile and load our kernel module and view the kernel ring buffer messages.

2. You want to see the last 20 lines, provide an option to tail as tail -20. Notice the different timestamps when Hello world! and Goodbye world! are printed (15pts).
   In the screenshot we ran dmesg -T | tail and dmesg -T | tail - 20 and you can see the differences below:

```
[root@localhost OS_P2]# insmod hello.ko
[root@localhost OS_P2]# dmesg -T | tail
[Thu Oct  5 13:25:23 2023] IPv6: ADDRCONF(NETDEV_UP)
[Thu Oct  5 13:25:24 2023] virbr0: port 1(virbr0-nic
[Thu Oct  5 13:25:58 2023] ISO 9660 Extensions: Micr
[Thu Oct  5 13:25:58 2023] ISO 9660 Extensions: RRIF
[Thu Oct  5 13:26:02 2023] TCP: lp registered
[Thu Oct  5 13:56:39 2023] hello: loading out-of-tre
[Thu Oct  5 13:56:39 2023] hello: module verificatic
[Thu Oct  5 13:56:39 2023] Hello world!
[Thu Oct  5 14:17:07 2023] Goodbye world!
[Thu Oct  5 14:37:20 2023] Hello world!
[root@localhost OS_P2]# dmesg -T | tail -20
[Thu Oct  5 13:25:18 2023] vmxnet3 0000:03:00.0 ens1
[Thu Oct  5 13:25:19 2023] nf_conntrack version 0.5.
[Thu Oct  5 13:25:19 2023] bridge: filtering via arp
.
[Thu Oct  5 13:25:23 2023] tun: Universal TUN/TAP de
[Thu Oct  5 13:25:23 2023] tun: (C) 1999-2004 Max Kr
[Thu Oct  5 13:25:23 2023] virbr0: port 1(virbr0-nic
[Thu Oct  5 13:25:23 2023] virbr0: port 1(virbr0-nic
[Thu Oct  5 13:25:23 2023] device virbr0-nic entered
[Thu Oct  5 13:25:23 2023] virbr0: port 1(virbr0-nic
[Thu Oct  5 13:25:23 2023] virbr0: port 1(virbr0-nic
[Thu Oct  5 13:25:23 2023] IPv6: ADDRCONF(NETDEV_UP)
[Thu Oct  5 13:25:24 2023] virbr0: port 1(virbr0-nic
[Thu Oct  5 13:25:58 2023] ISO 9660 Extensions: Micr
[Thu Oct  5 13:25:58 2023] ISO 9660 Extensions: RRIF
[Thu Oct  5 13:26:02 2023] TCP: lp registered
[Thu Oct  5 13:56:39 2023] hello: loading out-of-tre
[Thu Oct  5 13:56:39 2023] hello: module verificatic
[Thu Oct  5 13:56:39 2023] Hello world!
[Thu Oct  5 14:17:07 2023] Goodbye world!
[Thu Oct  5 14:37:20 2023] Hello world!
```

The “tail” command displays the last 10 lines. Where as the “tail -20” command displays the last 20 lines. Any number can be used after the tail command, for example if you wanted to see the last 100 lines you would use “tail -100”.

3. If I want to print the messages from init module and cleanup module to the standard output in addition to the kernel ring buffer, what should I change (5pts)?
   Since kernel modules cannot write to std out, you cannot utilize the stdio.h library to simply use some method like print() or printf(). If you want to print to the standard output in addition to the kernel ring buffer you would need to use the dmesg command. An example would be a command such as dmesg with -k option which will display kernel messages from the kernel ring buffer, filtering out any userspace messages.
   The -T displays timestamps while the -k displays kernel messages to stdout.

   In this screenshot we ran “dmesg -T | tail” and then ran “dmesg -k | tail”.

```
[Thu Oct  5 13:56:39 2023] Hello world!
[Thu Oct  5 14:17:07 2023] Goodbye world!
[Thu Oct  5 14:37:20 2023] Hello world!
[root@localhost OS_P2]# dmesg -k | tail
[   18.902377] IPv6: ADDRCONF(NETDEV_UP): virbr0: link is not ready
[   19.310828] virbr0: port 1(virbr0-nic) entered disabled state
[   53.588858] ISO 9660 Extensions: Microsoft Joliet Level 3
[   53.715887] ISO 9660 Extensions: RRIP_1991A
[   57.170931] TCP: lp registered
[ 1894.228547] hello: loading out-of-tree module taints kernel.
[ 1894.228747] hello: module verification failed: signature and/or required key
[ 1894.229368] Hello world!
[ 3122.029463] Goodbye world!
[ 4335.746885] Hello world!
[root@localhost OS_P2]#
```

Part 2:

For the second part of the assignment we were tasked with implementing a module print self. This module identifies the current process using the “current” macro and prints out and displays the messages from the kernel ring buffer. We needed the methods to Implement the print self module and print out the following:

- Process name, id, and state;
- The same above information of its parent processes until init.

For part 2 we created a different subdirectory and a new makefile. We decided to use the corrected Makefile that we made for part 1 and simply change the object file within the makefile to be that of print_self.o. We added linux library #include <linux/sched.h>

which is a library for process functions and allows use of struct task_struct. We used the “current” macro which is provided in the linux kernel libraries. By doing so, we were able to initialize the module with the process retrieved from the “current” and then display the required kernel messages for that process (i.e. Process name, id, and state). Afterwards, we utilized a while loop which traverses up the parent tree to print the same properties as we did for the process retrieved by the “current” macro for all parent processes up until systemd (PID 1).

In your report, please (1) list steps to load and remove your module, and read your module’s output; and (2) answer the following questions:

The command to load the module for part 2 is:
insmod print_self.ko
The command to read the modules output for part 2 is:
dmesg -T | tail -20
The command to remove the module for part 2 is:
rmmod print_self

1. The macro current returns a pointer to the task struct of the current running process. See the following link: https://linuxgazette.net/133/saha.html When you load your module, which process is recognized as current?
   Since we are loading the module using insmod, the process executing insmod is the "current" process. Therefore, the current macro during its initialization will refer to the task_struct of the insmod process.

2. As discussed in our lecture, in old kernels the mother of all processes is called init. In newer kernels, what is it called and what do you see from your module’s output?

   In old Kernels the mother of all processes is called “init”. In newer Kernels this process is called “systemd’. This is the new initialization framework that is more modern and contains more features than init. Systemd provides parallelization, service management, tracking, and centralized logging. It is becoming the standard init system for most Linux distributions. In our modules output, we can see the the process with PID of 1 is in fact systemd which corresponds to the fact the mother of all processes in newer kernels is systemd.

3. To see the different states of a process, please refer to the same page above https://linuxgazette.net/133/saha.html When printing state in your code, please map the numeric state to its string state, e.g., print TASK RUNNING if state is 0. From your module’s output, which state(s) are observed?

   In this screenshot below:
   We ran the command “dmesg | tail -5” this shows the numeric state to its string state for the module output for the last 5 lines. Process State: 0 is

TASK_RUNNING and Process State: 1 is TASK_INTERRUPTIBLE. If you want to see more you can expand the tail to be any value to show more tasks and the states string and numeric values.

```
[root@localhost part2]# dmesg | tail -5
[ 1103.716827] Parent Process ID: 2976
[ 1103.716829] Parent Process State: 0 (TASK_RUNNING)
[ 1103.716831] Parent Process Name: systemd
[ 1103.716833] Parent Process ID: 1
[ 1103.716835] Parent Process State: 1 (TASK_INTERRUPTIBLE)
```

Part 3:

For part 3 we created a different subdirectory and a new makefile which was nearly identical to that of parts 1 and 2. We changed the specified object file to print_other.o and introduced the ability to allow the makefile to get a random PID using grep and echo this retrieved PID into a newly created file called random_pid.txt which is all done by the makefile. The makefile utilizes grep to retrieve an arbitrary PID and stores it in a newly created txt file called random_pid.txt. We added the #include <linux/pid.h> library which provides the necessary functionality to be able to search for a process by its PID (in this case the arbitrary PID). We were able to implement our module such that when it is loaded, it searches and finds the process whos PID is passed in as an argument. When loading the module using the 'insmod' command, the kernel object (.ko) is passed into the command along with a parameter *target_pid=$(cat random_pid.txt)* which sets the target_pid in the module to the arbitrary PID that is held in the .txt file. We then display the required kernel messages for that process (i.e. Process name, id, and state). Afterwards, we utilized a while loop which traverses up the parent tree to print those same properties for all parent processes up until process systemd (PID 1).

In your report, please list steps to load and remove your module, and read your module's output.

The command to load the module for part 3 is:
insmod print_other.ko target_pid=$(cat random_pid.txt)
The command to read the modules output for part 3 is:
dmesg -T | tail -20
The command to remove the module for part 3 is:
rmmod print_other
NOTE: If you would like to run the module again and want it to use a different arbitrary PID, then after removing the module and running make clean, manually remove the random_pid.txt file and follow steps 1 - 3 again.

Part 4:

So what exactly is a kernel module? Please read this short article (kernel modules are also known as loadable modules). Use your own words, please answer the following:

A kernel module is code that is loaded or unloaded from the kernel at runtime. Allowing for the dynamic extension of the kernel's capabilities and support for various hardware devices and software features. (insmod for inserting and rmmod for removing).

1. What's the difference between a kernel module and a system call?

Kernel modules are a way to add functionality to the kernel without having to recompile the entire kernel. They can be loaded and unloaded. They're useful for device drivers, filesystems, or any feature that may be needed temporarily. They can be loaded into the kernel and unloaded from the kernel. They provide a way to extend the kernel's capabilities without rebooting the system.

System calls provide an interface through which user-space programs can request services (like I/O operations, process management, etc.) from the kernel. They are also unique from kernel modules in that they act as a bridge between the user-space and the kernel-space. System calls are built into the kernel. Unlike a kernel module, once a kernel is compiled with specific system calls, they can't be added or removed without modifying the kernel's code and recompiling it.

2. This article is over 20 years old. If you try this example from the article in your VM, Does it still work? Use your own words to explain why you think this may be a good (or bad) thing.

The code example provided is an attempt to hook a system call by directly modifying the sys_call_table. When we tried this in our VM, it did not work as intended. The modern Linux kernels have undergone significant changes over the past 20 years. For example, here are some reasons why this code may not work: The modern Linux kernel does not export the sys_call_table symbol, making it inaccessible from a kernel module. The code lacks some essential headers which are now mandatory for kernel modules. Considering the security implications of allowing modules to hook system calls by directly changing the system call table, it's clear that these kernel changes are for the better. Allowing such modifications could lead to security vulnerabilities, and unintended behaviors. While this method might have been feasible two decades ago, it's a good thing that it doesn't work now. It shows the kernel community's commitment to improving security and adapting to modern software practices.