

CS 4500 – (003) Operating Systems
Project 2: Kernel Modules
Drew Houchens
2/26/2026

1.0 Statement

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

2.0 Virtual Machine Information

Name of Virtual Machine: HouchensVM

Password for Instructor Account: Broncos4me05-

Path to Source Code: /home/drew/Operating-Systems/Project2

3.0 Description

2.1 How We Solved the Problems

When compiling the hello module in Part 1, I got a "redefinition of 'cleanup_module'" error. The sample code provided in the assignment defined functions named `init_module()` and `cleanup_module()` while also using the `module_init()` and `module_exit()` macros. In modern kernels, these macros internally create their own `init_module` and `cleanup_module` symbols, causing a naming conflict. The fix was to rename the functions to unique names (e.g., `hello_init` and `hello_exit`) and let the macros handle the mapping. This same fix was applied to the `print_self` and `print_other` modules.

In Part 3, after successfully building the `print_other` module, I repeatedly received errors when trying to load it with `insmod`. The issue was the PID I was passing as an argument was no longer valid. The processes had already terminated by the time the module loaded. Since `insmod` only shows a generic error message and does not display the module's `printk` output, I had to check `sudo dmesg -T | tail` to see the actual "No process found with PID" message from our module. Then, I used `pgrep bash` to get a valid PID and the module loaded correctly.

2.2 What We learned

I learned that kernel module development requires careful attention to how the kernel API evolves. Function names and macros that appear in older documentation or sample code may conflict with modern kernel internals. I also learned that kernel modules cannot print directly to the user's terminal. All output goes to the kernel ring buffer accessed via `dmesg`. This makes debugging less intuitive than user-space programs.

4.0 Report

Part 2: Create a `print_self` kernel module – Question Responses

Output when running:

```
sudo insmod print_self/print_self.ko
```

```
sudo dmesg -T | tail -20
```

```
tplan-ovs-cleanup.service is marked world-inaccessible. This has no effect as co
nfiguration data is accessible via APIs without restrictions. Proceeding anyway.
[Thu Feb 26 11:43:40 2026] hello: loading out-of-tree module taints kernel.
[Thu Feb 26 11:43:40 2026] hello: module verification failed: signature and/or r
equired key missing - tainting kernel
[Thu Feb 26 11:43:40 2026] Hello world!
[Thu Feb 26 11:46:00 2026] Goodbye world!
[Thu Feb 26 11:47:13 2026] print_self: Module loaded
[Thu Feb 26 11:47:13 2026] print_self: Walking process tree from current to init
[Thu Feb 26 11:47:13 2026] print_self: NAME          PID          STATE
[Thu Feb 26 11:47:13 2026] print_self: ----          ---          -----
[Thu Feb 26 11:47:13 2026] print_self: insmod          7159          TASK_RUNN
ING (0)
[Thu Feb 26 11:47:13 2026] print_self: sudo            7158          TASK_INTE
RRUPTIBLE (1)
[Thu Feb 26 11:47:13 2026] print_self: sudo            7157          TASK_INTE
RRUPTIBLE (1)
[Thu Feb 26 11:47:13 2026] print_self: bash            2496          TASK_INTE
RRUPTIBLE (1)
[Thu Feb 26 11:47:13 2026] print_self: gnome-terminal-  2478          TASK_INTE
RRUPTIBLE (1)
[Thu Feb 26 11:47:13 2026] print_self: systemd          1445          TASK_INTE
RRUPTIBLE (1)
[Thu Feb 26 11:47:13 2026] print_self: systemd           1          TASK_INTE
```

1. When loading the module, the process recognized as current is **insmod** (PID 7159). The old article references a header file `<asm/current.h>` that is no longer needed. In modern kernels, `#include <linux/sched.h>` is all that is required.
2. In older kernels the process with PID 1 was called **init**. In newer kernels it is called **systemd**. Our output shows systemd at PID 1 at the bottom of the parent chain.
3. The old article uses `task_struct->state` but this has been renamed to `task_struct->__state` in newer kernels. From our output we observed two states.
TASK_RUNNING (0) for the insmod process and **TASK_INTERRUPTIBLE** (1) for all parent processes.

Part 3: Create a print other kernel module – Question Responses

```

RRUPTIBLE (1)
[Thu Feb 26 11:48:14 2026] print_self: Module removed
[Thu Feb 26 11:48:58 2026] print_other: Module loaded with pid=7158
[Thu Feb 26 11:48:58 2026] print_other: No process found with PID 7158
[Thu Feb 26 11:55:24 2026] print_other: Module loaded with pid=7158
[Thu Feb 26 11:55:24 2026] print_other: No process found with PID 7158
[Thu Feb 26 11:57:31 2026] print_other: Module loaded with pid=7158
[Thu Feb 26 11:57:31 2026] print_other: No process found with PID 7158
[Thu Feb 26 11:58:16 2026] print_other: Module loaded with pid=7159
[Thu Feb 26 11:58:16 2026] print_other: No process found with PID 7159
[Thu Feb 26 11:58:45 2026] print_other: Module loaded with pid=2496
[Thu Feb 26 11:58:45 2026] print_other: Walking process tree from PID 2496 to in
it
[Thu Feb 26 11:58:45 2026] print_other: NAME                PID      STATE
[Thu Feb 26 11:58:45 2026] print_other: ----                -
[Thu Feb 26 11:58:45 2026] print_other: bash                2496     TASK_INT
ERRUPTIBLE (1)
[Thu Feb 26 11:58:45 2026] print_other: gnome-terminal-      2478     TASK_INT
ERRUPTIBLE (1)
[Thu Feb 26 11:58:45 2026] print_other: systemd              1445     TASK_INT
ERRUPTIBLE (1)
[Thu Feb 26 11:58:45 2026] print_other: systemd              1       TASK_INT
Help
drew@drew-virtual-machine:~/Operating-Systems/Project2$

```

(you can see me trying to find nonexistent processes)

1. Not all kernel functions are available to modules. I used the `for_each_process()` macro from `<linux/sched/signal.h>` to iterate through the process table and find the target process by PID. Functions like `find_task_by_vpid()` are not exported to modules.

2. I used `pgrep bash` to get a valid PID of a running bash shell. We got PID 2496 and passed it to the module.
3. I passed the PID as an argument using `module_param(pid, int, 0644)` in the code.
To load: `sudo insmod print_other/print_other.ko pid=2496`. To read output: `sudo dmesg -T | tail -20`. To remove: `sudo rmmod print_other`.

Part 4: Kernel Modules and System Calls – Question Responses

1. A kernel module is a piece of code that can be loaded into and removed from the kernel at runtime without rebooting. It extends what the kernel can do, such as adding a driver. A system call is a function that lets a user-space program request a service from the kernel. An example of this is requesting to read a file.
A module adds code to the kernel while a system call is an interface for programs to talk to the kernel.
2. The example from the article would not work on a modern kernel. Kernel internals like function names, struct fields, and exported symbols change over time. Even in this project, `task_struct->state` was renamed to `__state`. Old header files were removed and some functions are no longer exported to modules. This is a good thing because it means the kernel is improving and being maintained, but it also means old code goes out of date quickly.