Assignment #1
CS695 Spring 2023-24

linux: **L**inux **In**ternals **U**nderstanding and e**X**ploration

0. Setting up the environment

Submissions will be evaluated on Linux machines running kernel version **6.1.0** As kernel data structure changes with kernel versions, ensure your solution works for the mentioned version. You can use the `'uname -r'` command to check the kernel version in a Linux machine.

Setup Choices:

Have a Linux machine running kernel version **6.1.0** with root access. If you are running a native install, you should be able to do this without using a VM, but it is highly recommended to use a VM to avoid potentially corrupting your native system.

Setup a Virtual Machine (long live virtualization!)

You can download and use the following VM images as per your machine architecture to quickly get started with the assignment —

x86-64 (VirtualBox Image): link
arm64 (UTM Image): link

Install VirtualBox (or UTM if you have Apple Silicon)

VirtualBox: https://www.virtualbox.org/wiki/Downloads
UTM: https://mac.getutm.app/ (download from the website is free of cost)

Import and run the image.
Credentials:
username: *cs695*
password: *1234*
root password: *1234*

Pro Tip: Since the VM images do not have a GUI, you can set up an editor (VSCode, etc.) over ssh. ssh-server is enabled in the images by default, although for VirtualBox, you must change the Network Adapter mode to “Bridged Adapter” to let the host connect to the VM.

Warning: Some exercises involve tinkering with the kernel at a very low level. If anything goes wrong, your system may crash (or even get corrupted). It is better to do these exercises on a virtual machine.

1. Hello LKM!

A Linux kernel module (LKM) is a piece of code (kernel functionality) that can be loaded and unloaded into the kernel on demand. Kernel modules extend the kernel's functionality without rebooting the system. They are typically used for on-demand loading of drivers to handle plug-and-play devices and adding custom system calls like functionality to the kernel, etc.

To get started, ensure you are ready with the setup specified in *Section 0*. Run the following commands for installing packages on your system (Ubuntu/Debian) for building and installing kernel modules:

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

Follow the link https://linux-kernel-labs.github.io/refs/heads/master/labs/kernel_modules.html to get pointers on writing, loading, and unloading Linux kernel modules.

Implement a kernel module named **`helloworld.ko`**, which prints a "Hello World" when the module is loaded and an exit message "Module Unloaded" when the kernel is unloaded.

1.1. Peeking into Linux

Implement the following functionalities to peek into the Linux kernel implementation.

> Write a kernel module to list all processes in a running or runnable state. Print their pid. Name the source file as **`lkm1.c`**
>
> Write a kernel module that takes process ID as input, and for each of its child processes, print their pid and process state. Name the source file as **`lkm2.c`**
> Write a kernel module that takes process ID and a virtual address as its input. Determine if the virtual address is mapped, and if so, determine its physical address (pseudo physical address if running in a VM). Print the pid, virtual address, and corresponding physical address. Name the source file as **`lkm3.c`**
>
> For a specified process (via its pid), determine the size of the allocated virtual address space (sum of all vmas) and the mapped physical address space. Write a test program that allocates different sizes of memory in stages, and in each stage, observe the mapped memory stats using your LKM. Make sure that you allocate memory in the granularity of page size. Name the source file as **`lkm4.c`**

Note: Your submission should include a common Makefile and/or Kbuild file to compile all four modules.

The following is a good source for easily navigating through the kernel source code
Linux source code (v6.1) - Bootlin

Kernel structures you may need to understand and use —
`task_struct, mm_struct, vm_area_struct, maple_tree, mmap.h, mm.h`

Reference: Message logging from the kernel
Introducing maple trees [LWN.net]
Maple Tree — The Linux Kernel documentation

2. ioctl — *one system call to rule them all*

Familiarize yourself with the `ioctl` system call
Online resources:

- Input/Output Control in Linux | ioctl() implementation
- Talking to Device Files (writes and IOCTLs)
- Character device drivers — The Linux Kernel documentation
- GitHub - pokitoz/ioctl_driver: Example on how to write a Linux driver

As a warm up exercise, follow the online resources to write an ioctl driver that returns a constant to the caller. For this you will have to create a device file, register your custom ioctl functions and write a user space program to make the ioctl calls.

Captain Kirk and Science Officer Spock are curious about the virtual-to-physical address mappings within his spaceship object collision program. As modern Operating Systems do not allow user-space programs to break their memory abstraction, we will use ioctl calls to provide the required functionality.

Implement an ioctl device driver which provides the following functions:

Provide the physical address for a given virtual address. The physical to virtual-address translation should be done for the current running process.

Given a physical address and a byte value, it should write the value at the memory address.

Write a user space program to implement the following operations:

Allocate a byte-size memory on the heap.

Assign the value “6” to the memory.

Print the address (virtual address) and the value of the allocated memory.

Make an ioctl call to get the physical address of the allocated memory. Print the physical address.

Make another ioctl call to change the value of the memory to “5” using a physical memory address.

Verify the modified value by printing the content of the allocated memory.

Write a script **`spock.sh`,** which

compiles your ioctl driver and user space application

initiates the ioctl device

runs your user space application

cleanly removes the ioctl device

Note: Include any Makefile/Kbuild files necessary to build all your applications and ioctl driver. Invoke the build within **`spock.sh`** script.

Hint: All memory accesses are through virtual addresses, even in kernel mode. You should convert the physical address to a virtual address within the module.

Reference: Memory Mapping and DMA

> Dr. Doom is preparing to invade planet Earth. He has multiple soldiers ready to go to war for him and a central station that observes the entire operation. Whenever a soldier is killed or completes their task, the central station should be notified. Both soldiers and the central station are modeled using Linux processes. Remember that when a process exits, a **`SIGCHLD`** signal is delivered to its parent process. However, here, the central station and the soldiers are launched separately, so the station is not a parent to the soldiers. You agreed to help Dr. Doom by implementing an ioctl call to achieve the required functionality.

The central station program, the soldier program, and a script that launches these programs are available at this link.

Implement an ioctl device driver which provides a call with the following specifications:

Takes a pid as an argument

Modifies the task structure of the current process to change its parent process to the process with the given pid. More specifically, the process with the given pid should receive a **`SIGCHLD`** signal on exit of the current process (which makes the ioctl call).

Look for comments in the provided soldier program to make the ioctl call.

Modify the script at the mentioned places to compile, initiate, and remove your ioctl device.

Hint: Remember that in Linux when a process exits and if it has child processes, all child processes are inherited by the init process. You might look at how the parent is changed.

Reference: "parent" vs. "real_parent" in struct task_struct - Peilin Ye's blog
Iterating children of a task in task_struct

Kernel structures/functions of interest: `task_struct, do_exit in exit.c`

3. procfs — *file system with no files*

Let's take a look at something interesting in Linux called the /proc file system. It is a special kind of file system that helps interact with core parts of the operating system.

Now, what's cool about files in /proc is that they don't take up any real space on your computer. Instead, the kernel makes it look like there are files there, but the content inside is generated on the fly by the kernel.

Since the kernel module runs in a privileged mode and in the kernel address space, a module has access to all kernel data structures and effectively the whole system. The /proc file system thus provides an interface between the user space and a kernel module!

Note -
As many of the proc file system kernel functions are deprecated. Please use the following resources for kernel version >= 5.x.x to gain insights on utilizing the latest version functions.

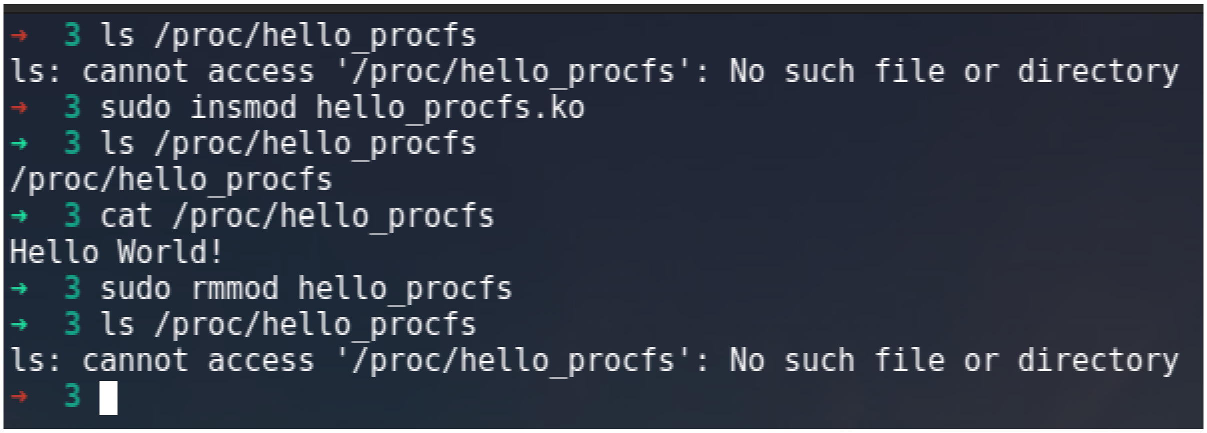
> The Linux Kernel Module Programming Guide -
> https://sysprog21.github.io/lkmpg/
>
> https://www.cs.fsu.edu/~cop4610t/lectures/project2/procfs_module/proc_module.pdf
>
> The seq_file Interface -
> https://www.kernel.org/doc/html/latest/filesystems/seq_file.html
> (advanced)

(a) Create a simple proc file entry

In this exercise, you will create a basic kernel module to introduce a new entry in the /proc filesystem.

Write a kernel module named `hello_procfs.c` that creates a new entry in the /proc filesystem called "`/proc/hello_procfs`". This entry should allow users to read and display "Hello World!" message when they use the "cat" command on it.

Usage -



```
➜  3 ls /proc/hello_procfs
ls: cannot access '/proc/hello_procfs': No such file or directory
➜  3 sudo insmod hello_procfs.ko
➜  3 ls /proc/hello_procfs
/proc/hello_procfs
➜  3 cat /proc/hello_procfs
Hello World!
➜  3 sudo rmmod hello_procfs
➜  3 ls /proc/hello_procfs
ls: cannot access '/proc/hello_procfs': No such file or directory
➜  3 █
```

Steps:

In the entry routine, create a new entry in the /proc directory named "`hello_procfs`".

Define your custom printf routine which is called when "`hello_procfs`" someone lists your proc file.

Define the open routine and substitute its name in `proc_ops` structure which is used to define the file manipulation callbacks for your proc file.

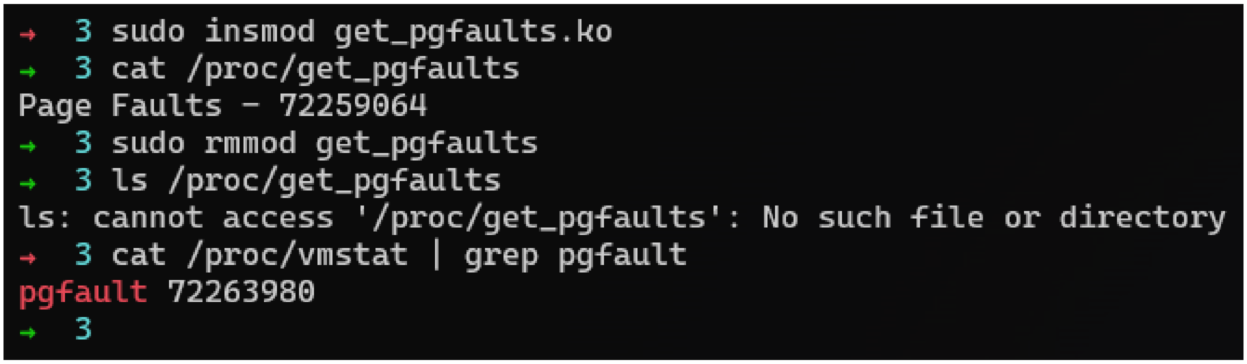
Remove the entry from the /proc directory in exit routine.

Hint: You might want to look at the following functions: proc_create, single_open, seq_printf remove_proc_entry.

(b) get page faults

Now, develop a kernel module named `get_pgfaults.c` with the objective of introducing a fresh entry into the /proc filesystem, specifically labeled "`/proc/get_pgfaults`". Users should be able to use commands like "cat" to retrieve and display information regarding the total count of page faults the operating system has encountered since it booted.

Usage:



```
→  3 sudo insmod get_pgfaults.ko
→  3 cat /proc/get_pgfaults
Page Faults - 72259064
→  3 sudo rmmod get_pgfaults
→  3 ls /proc/get_pgfaults
ls: cannot access '/proc/get_pgfaults': No such file or directory
→  3 cat /proc/vmstat | grep pgfault
pgfault 72263980
→  3
```

The Linux kernel already keeps track of page faults in a /proc file, along with many other stats. The following command given is a good example to compare the kernel stats with your module's stats.

```
cat /proc/vmstat | grep pgfault
```

Hint: You can refer https://elixir.bootlin.com/linux/v6.1/source to get an understanding of how to utilize `all_vm_events` function to get page faults.

Submission instructions

Your code should be well-commented and readable.
You need to submit solutions for 1.1, 2.I, 2.II and 3(a) and 3 (b)

All submissions have to be done via Moodle. Name your submission folder as well as the tarball as `<rollnumber>_assignment1.tar.gz` (e.g. `22d0371_assignment1.tar.gz`)

Please strictly adhere to this format otherwise, your submission will not count.

You can create the tarball using
`tar -czvf <rollnumber>_assignment1.tar.gz <rollnumber>_assignment1`

The tar should contain the following files in the following directory structure:

```
<rollnumber>_assignment1/
     ├──── 1/
     │        ├── lkm1.c
     │        ├── lkm2.c
     │        ├── lkm3.c
     │        ├── lkm4.c
     │        ├── Makefile
     │        └── Kbuild* <include only if using>
     │
     ├──── 2_I/
     │        ├── spock.sh
     │        └── ... <solution specific files/dirs>
     │
     ├──── 2_II/
     │        ├── control_station.c
     │        ├── soldier.c
     │        ├── run_dr_doom.sh
     │        ├── Makefile
     │        └── ... <solution specific files/dirs>
     │
     └──── 3/
          ├── hello_procfs.c
          ├── get_pgfaults.c
          ├── Makefile
          └── Kbuild* <include only if needed>
```

Deadline: 29th January 2024, 11:59 PM via Moodle.