cs695-assignment1

Automatisch alle 5 Minuten aktualisiert

Assignment #1
CS695 Spring 2024-25

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.x-x** 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. [x in the Linux kernel version can be anything]

Setup Choices:

a. Have a Linux machine running kernel version **6.1.x-x** with root access. If you are running a native install, you should be able to do this without using a VM.

b. Setup a Virtual Machine (long live virtualization!)

   i. 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

   ii. Install VirtualBox

      VirtualBox: https://www.virtualbox.org/wiki/Downloads

   iii. 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 make the following changes:
      1. Go to Devices > Network > Network Settings. Change the Network Adapter mode to "Bridged Adapter" to let the host connect to the VM.
      2. After changing the network adapter, reboot your virtual machine to make sure new IP addresses are allocated to you by the DHCP server. [Your host machine and VM will have different IP addresses]
      3. You have to log in to both machines to access the internet in the IITB network. A script `./internet.sh` has already been provided in the VM image for the same purpose; you may also find the script here. You should use LDAP user ID and Internet Access Token as password [link] for the script.
      4. Now, you can connect to the VM using cs695@<vm-ip>. You may use the same for your editor setup.

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

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.

Task 1.0. Implement a kernel module named `helloworld.ko`, which prints a message "Hello World" when the module is loaded and an exit message "Module Unloaded" when the kernel module is unloaded.

1.1. Peeking into Linux

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

Task 1.1 Write a kernel module to list all processes in a running or runnable state. Print their pid and process name. Name the source file as `lkm1.c`

Sample Message Log:

```
[ 8740.104553] [LKM1] Runnable Processes:
[ 8740.104557] [LKM1] PID       PROC
[ 8740.104558] [LKM1] ------------------
[ 8740.104568] [LKM1] 3750      cs695
```

```
[ 8740.104570] [LKM1] 3948      insmod
[ 8753.157720] [LKM1] Module LKM1 Unloaded
```

Task 1.2 Write a kernel module that takes process ID as input [input should be named **`pid`**] and prints their pid and process state for each of its child processes. Name the source file as **`lkm2.c`**

Sample Message Log:

```
[ 9270.167700] [LKM2] Child Process PID: 208, State: TASK_INTERRUPTIBLE
[ 9270.167704] [LKM2] Child Process PID: 237, State: TASK_INTERRUPTIBLE
[ 9270.167705] [LKM2] Child Process PID: 245, State: TASK_INTERRUPTIBLE
[ 9270.167707] [LKM2] Child Process PID: 474, State: TASK_INTERRUPTIBLE
[ 9270.167708] [LKM2] Child Process PID: 552, State: TASK_INTERRUPTIBLE
[ 9270.167709] [LKM2] Child Process PID: 4985, State: TASK_RUNNING
[ 9290.487710] [LKM2] Module LKM2 Unloaded
```

Task 1.3 Write a kernel module that takes process ID [input should be named **`pid`**] and a virtual address (in decimal) [input should be named **`vaddr`**] 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). In case of an unmapped address, proper messages should be printed otherwise, print the PID, virtual address, and corresponding physical address. Name the source file as **`lkm3.c`**

Sample Message Log:

```
[ 9898.276579] [LKM3] Virtual address: 0x5640f5a4f6b0 / 94836999124656
[ 9898.276582] [LKM3] Physical address: 0x36eca6b0 / 921478832
[ 9932.171724] [LKM3] Module LKM3 Unloaded
[ 9943.015934] [LKM3] VA not mapped: <reason?>
[ 9963.314022] [LKM3] Module LKM3 Unloaded
```

Task 1.4 For a specified process (via its PID) [input should be named **`pid`**], determine the size of the allocated virtual address space (sum of all vmas) and the mapped physical address space. Write a test program **`test1.c`**, that takes the number of pages to be allocated and a stride as CLI arguments. The program should first allocate the required pages and start from the first allocated address, write using that address, and continue writing at the next address, which is stride bytes away from the current address till the end of the allocated region. For example, an allocation of 128MB and a stride of 16MB will result in 8 writes.
Observe the mapped memory stats using your LKM for different combinations of the input tuple. Note that allocation of memory and accessing memory are two different actions. Write a short note on your observation in **`lkm4.txt`** and draw a graph with allocated pages on the x-axis, the mapped physical memory size of the process for different stride values. Make sure that you allocate memory in the granularity of page size. Name the source file as **`lkm4.c`** and **`lkm4.png`** for the graph.

Sample Message Log:

```
[10410.354845] [LKM4] Virtual Memory Size: 10656 KiB
[10410.354895] [LKM4] Physical Memory Size: 1512 KiB
[10426.035585] [LKM4] Module LKM4 Unloaded
[10667.045042] [LKM4] Virtual Memory Size: 10656 KiB
[10667.045068] [LKM4] Physical Memory Size: 3548 KiB
[10830.091668] [LKM4] Module LKM4 Unloaded
```

Task 1.5 In the Linux Kernel, the page size is usually 4K, but applications dealing with large memory working sets usually use huge page sizes (2M, 1G [may vary with architecture]) to optimize its performance. One way to use hugepages is to allocate anonymous memory and enable Transparent HugePage Support (THP) to promote and demote pages to different sizes automatically. Write a test program **`test2.c`**, that allocates anonymous memory of reasonably large size. For the specified process (via its PID) [input should be named pid], determine the number of huge pages and the allocated virtual address space they use when THP is enabled and when it is disabled. The source file should be named **`lkm5.c`**

Sample Message Log:

```
[114958.052158] [LKM5] THP Size: 202752 KiB, THP count: 99
[114970.326244] [LKM5] Module LKM5 Unloaded
[115056.042726] [LKM5] THP Size: 0 KiB, THP count: 0
[115139.727883] [LKM5] Module LKM5 Unloaded
```

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

The following is a good source for easily navigating through the linux 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`, `etc.`

Reference:
Message logging from the kernel
Linux Process States
Introducing maple trees [LWN.net]
Maple Tree — The Linux Kernel documentation
Transparent Hugepage Support — Usecase documentation
Transparent Hugepage Support — The Linux Kernel documentation

But why are we learning about Linux kernel modules in this course? As it turns out, to run other operating systems in virtual machines alongside your host operating system, the VM needs to control your processor and other hardware. It does so by using kernel modules. Use `lsmod` in your host OS to check the loaded modules. What kernel modules do you think virtualbox uses and why? 🤔

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.

Task 2.1 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.

A. Implement an ioctl device driver which provides the following functions:

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

II. Given a list of **physical** addresses and corresponding byte values, write the specified byte value to each of the provided physical memory addresses. The physical address and the value to be written as parameters of the **ioctl** call.
For example,
User space processes uses the ioctl call to translate a virtual address **X** to its physical mapping **Y**.
Next, it uses another ioctl call to issue a write using address **X** and some value **B**.

B. Write a user space program to implement the following operations:

I. Allocates a memory address block of size ***count*** bytes on the heap

II. Assign the value "104" to the first byte of the allocated memory address

III. Assign a value, Incremented by 1, for each consecutive memory address until the entire block is filled.

IV. Print the addresses (virtual addresses) of the allocated memory and their corresponding values.

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

VI. Implement another ioctl call to update the value at the first memory address to 53 and increment the values at subsequent addresses by 1, using their respective physical memory addresses

VII. Verify the modified values by printing the content of the allocated memory block.

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

I. compiles your ioctl driver and user space application

II. initialize the ioctl device

III. runs your user space application

IV. 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.

Sample Output:

```
cs695@cs695:~$ ./spock.sh
VA: 0x563add6c42a0 Value: 104
VA: 0x563add6c42a1 Value: 105
VA to PA translation
VA: 0x563add6c42a0        PA: 0x24d962a0
VA: 0x563add6c42a1        PA: 0x24d962a1
PA: 0x24d962a0        Value: 104
PA: 0x24d962a1        Value: 105
VA: 0x563add6c42a0        PA: 0x24d962a0        Updated Value: 53
VA: 0x563add6c42a1        PA: 0x24d962a1        Updated Value: 54
```

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

Reference: Memory Mapping and DMA

Task 2.2 Dr. Bloom is preparing to heal planet Earth. He has multiple foot soldiers ready to land to Earth for the task and also a central control station that observes the entire operation. Whenever a foot soldier completes the assigned task, the central station should be notified. Both the foot 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 foot soldiers are launched separately, so the station is not a parent

process to the soldier process. We as earthlings have agreed to help Dr. Bloom by implementing ioctl calls to achieve the following functionality.

A. Implement the following **ioctl** device driver which provides a call with the following specifications:

   a. Takes a **pid** as an argument and modifies the task structure of the current process to change its parent 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).

      This IOCTL call/functionality is used by the foot soldier program to attach itself to the control station.

   b. A process use **pid** as an argument and terminates all child processes of the given pid. After all child processes get terminated, the process terminates/exits itself. This ioctl call is done on an emergency

      This second IOCTL call/functionality is used by the control station to initiate a reset condition to stop/terminate the foot soldier programs.

B. The soldier and control station programs to invoke the ioctl calls, and a script that launches these programs are provided at this link.

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

Sample Output:

```
cs695@cs695:~$ ./run_dr_bloom.sh
Control station PID: 12148
Soldier PID: 12149
Soldier PID: 12150
Soldier PID: 12151
Soldier PID: 12152
[CHILD]: soldier 12152 changing its parent
[CHILD]: soldier 12151 changing its parent
[CHILD]: soldier 12150 changing its parent
[CHILD]: soldier 12149 changing its parent
[PARENT]: Control station process 12148 started
[PARENT]: Soldier process 12150 terminated
[PARENT]: Soldier process 12149 terminated
[PARENT]: Emergency Emergency!
[PARENT]: Soldier process 12152 terminated
[PARENT]: Soldier process 12151 terminated
[PARENT]: Control station 12148 exiting
```

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 and sysfs — *file systems with no files*

Next, we will study the **procfs** and **sysfs** file systems of Linux. These file systems provide a file interface (via the **/proc** and **/sys** directories) for interaction with the kernel during execution.

The interesting part about files in `/proc` and `/sys` is that they do not take up any real disk space. Instead, the Linux kernel uses the VFS (virtual file system) design to implement the file functions to generate content on the fly based on kernel state and to update kernel state.

Until now, we were using `printk()` or/and its variants to display messages to the kernel log buffer, which were then read using `dmesg`. The `/proc` file system is an additional mechanism in Linux where the kernel and kernel modules can send information to userspace processes. While `procfs` is generally used for reporting information from kernel and/or kernel modules. `sysfs` allows interaction with the running kernel from user space by reading or setting variables inside modules using a file operations.

Note:
Many of the procfs and sysfs kernel functions are deprecated. Please use the following resources for kernel version >= 5.x.x to gain insights on utilizing the latest version functions.

procsfs and sysfs guide - The Linux Kernel Module Programming Guide
Using procfs with Linux Kernel Modules - procfs Kernel Module
Using sysfs with Linux Kernel Modules - A complete guide to sysfs - Part 1

Advanced Topics
- The seq_file Interface — The Linux Kernel documentation
- A complete guide to sysfs — Part 2: improving the attributes
- A complete guide to sysfs — Part 3: using kset on kobject

Task 3.0.1 In this exercise, we 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 a `Hello World!` message when the file is read.

Task 3.0.2 Similar to the previous exercise, we will create another kernel module that creates a new entry to the `/sys` filesystem. Name the kernel module `hello_sysfs.c`, it should create a directory in the `/sys` filesystem called `/sys/kernel/hello_sysfs`, which has two files `hello_int` which represents an integer and `hello_string` which represents a string. When

the kernel module is compiled and loaded, you should be able to use echo and cat commands to store and show the value in the files.

Task 3.1 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.

Sample Usage:

```
cs695@cs695:~$ sudo insmod get_pgfaults.ko
cs695@cs695:~$ cat /proc/get_pgfaults
Page faults: 4382742
cs695@cs695:~$ sudo rmmod get_pgfaults
cs695@cs695:~$ cat /proc/get_pgfaults
cat: /proc/get_pgfaults: No such file or directory
cs695@cs695:~$ cat /proc/vmstat | grep pgfault
pgfault 4398679
cs695@cs695:~$
```

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

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

Hint: Refer to `all_vm_events` function of linux kernel.

Task 3.2 Write a kernel module `get_memstats.c` that creates a `/sys/kernel/mem_stats` directory to expose memory statistics similar to **Task 1.4.** The directory should have the following four files:

1. **pid**: The pid of the process for which the memory statistics should be shown. The file should have both read and write and access and contain a default value of -1.
2. **virtmem**: The current virtual memory size of the process represented by the pid. This file should only have read access and, by default, should show the values in Bytes.
3. **physmem**: The current physical memory of the process represented by the pid. Similar to virtmem the file should also be read-only, and by default should show the values in Bytes.
4. **unit**: Unit of memory that should be shown in the virtmem and physmem files. This file is both readable and writable. The file can take three inputs “B” representing Bytes, “K” representing KB, and “M” representing MB. On kernel module loading by default the file should contain “B” and should be changeable at runtime. All other values should be discarded appropriately.

Note that If pid is absent (i.e. pid should contain -1), the virtmem and physmem files should also show -1.

Sample Usage:

```
cs695@cs695:~$ sudo insmod get_memstats.ko
cs695@cs695:~$ ls /sys/kernel/mem_stats/
physmem pid unit virtmem
cs695@cs695:~$ cat /sys/kernel/mem_stats/unit
B
cs695@cs695:~$ cat /sys/kernel/mem_stats/pid
-1
cs695@cs695:~$ echo 518 > /sys/kernel/mem_stats/pid
cs695@cs695:~$ cat /sys/kernel/mem_stats/virtmem
10911744
cs695@cs695:~$ cat /sys/kernel/mem_stats/physmem
1548288
cs695@cs695:~$ echo M > /sys/kernel/mem_stats/unit
cs695@cs695:~$ cat /sys/kernel/mem_stats/virtmem
10656
cs695@cs695:~$ cat /sys/kernel/mem_stats/physmem
1512
```

Submission instructions

- Your code should be well-commented and readable.
- You need to submit solutions for 1.1, 2.1, 2.2, 3.1, and 3.2
- All submissions have to be done via Moodle. Name your submission folder as well as the tarball as `<rollnumber>_assignment1.tar.gz`
  (e.g. `24d0999_assignment1.tar.gz`) - all small letters

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

  ~~`24d0999_assignment1.gz`~~
  ~~`24d0999_assignment1.zip`~~
  ~~`24d0999_assignment1.tar.xz`~~

- 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
      │         ├── lkm4.png
      │         ├── lkm4.txt
      │         ├── lkm5.c
      │         ├── test1.c
      │         ├── test2.c
```

```
│       ├── Makefile
│       │       ├── README.txt* <if you include any
documentation>
│       └── Kbuild* <include only if using>
│
├─── 2.1/
│       ├── spock.sh
│       │       ├── README.txt* <if you include any
documentation>
│       └── ... <solution specific files/dirs>
│
├─── 2.2/
│       ├── control_station.c
│       ├── soldier.c
│       ├── run_dr_bloom.sh
│       ├── Makefile
│       │       ├── README.txt* <if you include any
documentation>
│       └── ... <solution specific files/dirs>
│
├─── 3.1/
│       ├── get_pgfaults.c
│       ├── Makefile
│       │       ├── README.txt* <if you include any
documentation>
│       └── ... <solution specific files/dirs>
│
└─── 3.2/
      ├── get_memstats.c
      ├── Makefile
      ├── README.txt* <if you include any documentation>
      └── ... <solution specific files/dirs>
```

Deadline: 25th January 2025, 11:59 PM via Moodle. (no extensions)