CS342 Operating Systems - Spring 2018
Project 3: Linux Kernel Modules and Process Memory

Assigned: April 13, 2018
Due date: April 30, 2018, 23:55

Objective*: Touching to the Linux kernel, learning Linux **kernel module development**, learning and getting experience with virtual memory, learning the virtual memory layout of a process, accessing page table.*

You will do this project in groups of 2 students. You can do it individually as well if you wish. You will do a lot of reading and thinking, but less coding in this project. You will do the project in a machine or better a virtual machine that has **64 bit Linux**, **Ubuntu 16.04**. The CPU architecture of the machine must be Intel **x86_64** or a compatible one.

In this project, you will develop a Linux kernel module and use it to see the memory usage of an applicaton. You will do the project in the following steps.

Step 1. Learn How to Write a Kernel Module

In this step you will learn how to develop and run (load/insert) a new kernel module. Compiling a module and loading and running it is quite easy and fast (just a few seconds) after you have the right development environment set up. There is documentation available on the web about Linux kernel module programming. Search for "Linux kernel module programming". At the end of this specification, there are also two good references to start with **[1,2**]. They can be reached from the course website. Read this documentation and do some simple exercises.

Step 2. Develop a Kernel Module

In this step, you will implement a Linux kernel module to get and print memory management information for a process. Your module will retrieve the required information from the related kernel data structures: the PCB of the process, the memory management data structure of the process, and the top level page table of the process. Your module will be a kernel code that can be loaded (inserted) and unloaded (removed) while the system (kernel) is up and running, without rebooting the system. It will be loaded with the **insmod** command. The module will take one argument, a process identifier (**pid** - an integer value), while being loaded using insmod. When loaded, a kernel module becomes part of the running kernel and runs in kernel mode and space with kernel priviliges.

Below are the things that your module will do.

a) It will first find the PCB of the process whose pid is specified at the command line while inserting the module. For that, your module can traverse the process list (PCB list). Depending on the kernel version, there may be a "current" variable in Linux kernel that is pointing to the PCB (of type tast_struct *) of the

currently running (scheduled) process. Starting from “current”, you can traverse the list of PCBs until you find the PCB of the process with the given pid. The PCBs in ready and running states are linked together (double linked list) in Linux kernel. There can be some other ways to traverse the process list; you can learn from Internet.

b) After finding the PCB, your module will print some basic memory management related information about the process. This information can be found by followinge the mm field (of type struct mm_struct *) of the PCB of the process. Basically, your module will print information about the virtual memory layout (the virtual memory regions) of the process. You know from lectures about the virtual memory of a process (seel Figure 1).

Figure 1. Virtual memory layout of a process.

The virtual memory layout of a process is a list of virtual memory regions (areas). Such regions together constitite the *virtual address space* of the process. A valid virtual address referenced by a running program must fall in one of those regions, i.e., must be in the virtual address space of the process. For each virtual memory area you need to print the following information in a line.

vm-area-start vm-area-end vm-area-size

Besides writing information about virtual memory regions of a process, you can print some additional information like the number of physical frames used by the process at that time, etc.

The information that will be printed out must include at least the following:

- start (virtual address), end (virtual address), and size of the *code*
- start, end, size of the *data*
- start, end, size of the *stack*
- start, end, size of the *heap*
- start, end, size of the main arguments,
- start, end, size of the environment variables
- number of frames used by the process (rss)
- total virtual memory used by the process (total_vm)

The printing will be done using the printk() kernel function. You can not use printf in kernel. The output will go to a kernel log file (in /var/log/…) that can be examined later by using commands like **dmesg**, more, cat, tail, etc.

You can verify your results by using the output of the following tools and commands:

- top
- ps aux
- cat /proc/pid/**maps**
- cat /proc/**meminfo**
- cat /proc/**vmstat**
- cat /proc/zoneinfo

You can verify your results with the output you can obtain from the /proc file system. Go into directory /proc. There you will see folders with integer names. Those integers are process ids. Change into a folder that is named with a pid (for example, the pid of the app process). Type 'more' or 'cat'. There you will see some files. They are actually virtual/special files whose content is not sitting on disk. The content for these files is obtained from memory-resident kernel structures. Type 'cat maps', for example to see the virtual memory regions of a process. Hence the /proc file system is a special file system corresponding to the kernel state. If you want to learn information about the kernel state, you can change into this /proc directory, traverse and obtain kernel state information. It is the interface of the kernel state to the users to learn about the values of some kernel variables and structures. The name /proc file system comes from 'process file system'.

c) Print the top-level page table content of the process. Parse each entry and write the information nicely. Since you will do your project in a 64 bit machine and you will use 64 bit Linux, 4 level page tables will be used. Assuming your machine architecture is Intel x86_64, a virtual address is 48 bits long and address division scheme is as follows: [9, 9, 9, 9, 12]. That means offset is 12 bits. The top level table has 2^9 = 512 entries. You can learn more about the page table structure of x86_64 architecture from Internet. Note that this part requires architecture knowledge (structure of the page table and page table entry the machine is using, etc.).

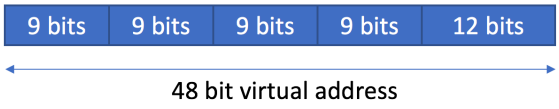

Figure 2. Virtual address format in x86_64 Intel architecture. The first -left most- 12 bits (high order bits) of a 64-bit address are all 0s. That means virtual addresses are actually 64 bits long, but the first 12 bits are not used, therefore they are effectively 48 bits long.

Step 3. Write an Application Allocating Memory Dynamically

Write an application app.c that will allocate and deallocate memory dynamically. Run this application, and while the application is running and allocating and deallocating memory, get VM usage information about this application using your module (insert module, remove module and check the kernel log file). Do such controlled experiments. For example, dynamically allocate memory using malloc() of some sizes and meanwhile see with your module how much the heap segment in the virtual memory of your process has extended. Call a function recursively and see

how stack extends. Compare the amount of allocation with malloc and the amount of extension in the heap section of the app process. Plot some graphics.

Report and Submission

You will write a report at the end. You will write in your report how you implemented your module, app, how you tested it and experimented with it, which information you are printing out, etc. Sample outputs will be included in the report. Include your module and app code as well.

You will upload your project (report, module, app, README.txt, etc.) to Moodle. Make sure you include the names of the group members in your report. One submission per group is enough.

You may be asked for a demo of your module. Then you will bring your machine and demonstrate your programs. It is also possible that we can do oral or written exams. Make sure each group member is working and learning. We can also have questions from the project in the exam.

[[Step 0. Building a New Linux Kernel

Do this step if you could not success in step 1 above. If you succeeded in step 1, you don't need to do this step. If you could not succeed with step 1 above, then doing this step 0 first and then doing the step 1 may help. Most of the time, you will not need this. In this step 0, learn how to compile (build) and run a new Linux kernel, so that you can get prepared to write a kernel module. Learn from Internet how to build and run a new kernel. Download source code of Linux kernel, build it (this may take a while – one hour or so – the first time you do it) and run it. You can do it on a virtual machine if you wish. If you are doing it directly on your machine, make sure **you backup all your data** first, so that if you mess up the file system and partitions on the disk, you can recover your data. Explain in your report briefly how you built and run your kernel. Note that this part will be quite time consuming, but you will do it only once (until you get your new kernel running).

You don't have to use the latest kernel version if you could not build it. You can use a recent but earlier version as well. You can download the source code of your existing/running kernel version as well. For that type "uname -r" to find out the version of your running kernel at the moment. Then you can download its source code from kernel.org and try to build that. In this case you can more comfortably use the kernel configuration file (config....) of your running kernel sitting in your /boot directory. **]]**

Optional Additional Information

[*optional info*] Note that the Linux kernel is also a C code that is compiled and that is running in the CPU. Hence it is also using virtual addresses. It can not use physical addresses. That means when kernel code runs in the CPU and makes a memory reference (generates a memory address), it is a virtual address. That virtual address is translated to a physical address using the page table of the process that made the system call and made the kernel running in the CPU. Hence a portion of the page table of a process is used to map kernel virtual addresses in Linux.

[*optional info*] Note that the kernel can not use a physical address or a physical frame number directly. To access a physical frame (given a frame number), the kernel first obtains a virtual address corresponding to the beginning of the frame using the kmap() function. Lets say, we want to access frame 34891. That means, frame number of the frame we want to access is 34891. We use kmap(mem_map + 34891) to obtain a corresponding virtual address. Then we can use that virtual address to access the frame content. For example, we can copy the frame content to

the application buffer by using that virtual address as the start virtual address of the frame to copy.

Clarifications

- The printk() function requires special formatting (%u, etc.) for printing unsigned integers.
- In kernel code you can not use printf(), since kernel is not linked with the standard C library.
- There are various places that tells you about the page table details of x86_64 architecture. Intel architecture manuals (that can be found on the web) can be used as well.

References

1. *The Linux Kernel Module Programming Guide*. http://www.tldp.org/LDP/lkmpg/2.6/html/index.html.
2. *Linux Device Dri*vers, Third Edition, http://lwn.net/Kernel/LDD3/, (especially the Chapter 2: Building and Running Modules).
3. *Linux Virtual Memory Manager*: https://www.kernel.org/doc/gorman/pdf/understand.pdf.
4. http://venkateshabbarapu.blogspot.com.tr/2012/09/process-segments-and-vma.html.
5. https://manybutfinite.com/post/anatomy-of-a-program-in-memory/.
6. https://manybutfinite.com/post/how-the-kernel-manages-your-memory/.
7. https://www.cse.iitk.ac.in/users/deba/cs698z/resources/linux-memory-2.pdf.
8. https://lwn.net/Articles/253361/.
9. http://www.tldp.org/LDP/tlk/kernel/processes.html.
10. Linux Cross Reference.
11. The Linux Unix Source Tour.
12. Building a Linux Kernel.