M1522.000800 System Programming, Fall 2018
L2.Kernel Lab: Linux Module Programming
Assigned: Mon., October 1, Due: Mon., October 15, 23:59

Introduction

This assignment will help you understand *Linux Kernel Module Programming* based on *Debug File System* interface. It has two assignments, the one is tracing process tree from specific process id, and the other is finding physical address using virtual address.

Note: The purpose of this lab is to learn *Linux Kernel Programming* and understand the difference between kernel-level programming and user-level programming. By this lab, you can use kernel data structures and understand the processing mechanism of them.

Background

Loadable Kernel Module

Loadable Kernel Module is an object file which contains kernel code to extend the functionality of kernel. There are two programmatic ways to access kernel space. The one is *System Call* which a program requests a service from the kernel of operating system. For adding *System Call* to the system, the kernel needs to be recompiled after the new *System Call* is enrolled to the system. It spends a lot of time to compile the whole kernel. The second is *Loadable Kernel Module* which can load and unload easily without the kernel compile. The developers can easily add the interface to access the kernel using *Loadable Kernel Module*.

Debug File System

Debug File System(*debugfs*) is special file system available in the Linux Kernel. *Debugfs* is a simple-to-use RAM-based file system specially designed for debugging purpose. It exists as a simple way for kernel developers to make information available to user space. Because *debugfs* has no rules at all, the developers can put any information they want. *Debugfs* also supports simple user-to-kernel interfaces in *Linux Kernel Module*. So, the user space developers can access kernel information easily using *debugfs*.

Linux Kernel Module Convention

The kernel developers have to follow the convention for *Linux Kernel Module*. See the source code below:

```
#include <linux/module.h>

MODULE_LICENSE("GPL");

static int __init init_my_module(void)
{
        // Running when this module is inserted to system
}

static void __exit exit_my_module(void)
{
        // Running when this module is removed from system
}

module_init(init_my_module);
module_exit(exit_my_module);
```

This is a basic frame of code for *Linux Kernel Module*. There are two functions, the one is called when the kernel module is inserted to system and the other is called when the kernel module is removed from system. The two functions are enrolled to the kernel using module_init and module_exit functions. MODULE_LICENSE macro declares which license the module uses.

Debugfs APIs

Linux Kernel offers some APIs for developers to use debugfs easily. Before seeing the *debugfs* APIs, we have to know how to connect the functions to the file operations interfaces in *Linux Kernel*.

```
#include <linux/fs.h>

static int open_op(struct inode *node, struct file *fp)
{
     // Running when open file operation is called
}

static int release_op(struct inot *node, struct file *fp)
{
     // Running when close file operation is called
}

static ssize_t write_op(struct file *fp,
                        const char __user *user_buffer,
                        size_t length,
                        loff_t *position)
```

```
{
     // Running when write file operation is called
}

static ssize_t read_op(struct file *fp,
                      char __user *user_buffer,
                      size_t length,
                      loff_t *position)
{
     // Running when read file operation is called
}

static const struct file_operations my_fops = {
     .open = open_op,
     .release = release_op,
     .write = write_op,
     .read = read_op,
};
```

file_operation structure offers file operations interfaces to developer. Developer implements file operations following the convention of the functions.

Now, let's see the *debugfs* APIs. There are some *debugfs* APIs based on file operations like below:

```
struct dentry *debugfs_create_dir(const char *name, struct dentry *parent)
struct dentry *debugfs_create_file(const char *name, umode_t mode,
                                struct dentry *parent, void *data
                                const struct file_operations *fops)

struct dentry *debugfs_create_u32(const char *name, umode_t mode,
                                struct dentry *parent, u32 *value)
struct dentry *debugfs_create_u64(const char *name, umode_t mode,
                                struct dentry *parent, u64 *value)

struct dentry *debugfs_create_x32(const char *name, umode_t mode,
                                struct dentry *parent, u32 *value)
struct dentry *debugfs_create_x64(const char *name, umode_t mode,
                                struct dentry *parent, u64 *value)

struct debugfs_blob_wrapper {
        void *data,
        unsigned long size;
};
struct dentry *debugfs_create_blob(const char *name, umode_t mode,
                                sturct dentry *parent,
                                struct debugfs_blob_wrapper *blob)
```

Hand Out Instructions

You can obtain the skeleton code for the Kernel Lab at:

```
https://git.csap.snu.ac.kr/
```

You can pull the project includes skeleton code for ***Kernel Lab*** from `SysProg/Template`. Follow the command below at your local repository.

```
unix> git remote add upstream https://git.csap.snu.ac.kr/Template/SysProg.git
unix> git pull upstream master --allow-unrelated-histories
```

After pull the repository, you can see the project structure under `SysProg/Lab/KernelLab` like this:

```
KernelLab
├─ptree
│ ├─dbfs_ptree.c
│ └─Makefile
└─paddr
  ├─app.c
  ├─dbfs_paddr.c
  └─Makefile
```

`ptree` directory is for assignment 1: *Process Tree Tracing*. It has skeleton C code(`dbfs_ptree.c`) and build script(`Makefile`). `paddr` directory is for assignment 2: *Find Physical Address*. It also has skeleton C code(`dbfs_paddr.c`) and build script(`Makefile`).

Hand In Instructions

You should submit below files for submissions:

1. Source code both Assignment 1 and 2
2. Makefile both Assignment 1 and 2
3. Report (PDF format)

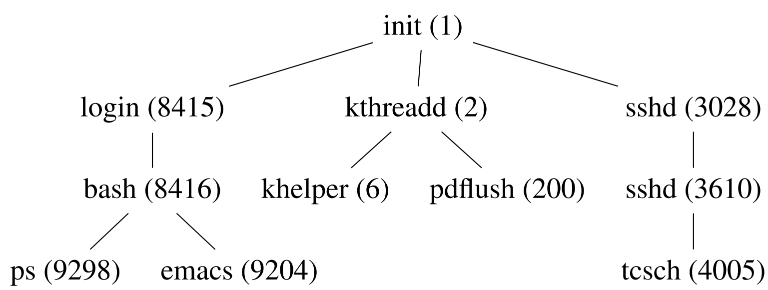
The report should follow strictly the uploaded report template on eTL. In the report, you describe the goal of Kernel Lab and how to implement for each Assignment. In conclusion section, you explain what you learn in this lab, what was difficult and what was surprising, and so on. You push all of your submissions(include the report) to your own git project for Kernel Lab. The project structure after submission is like below:

```
KernelLab
├── ptree
│   ├── dbfs_ptree.c
│   └── Makefile
├── paddr
│   ├── app.c
│   ├── dbfs_paddr.c
│   └── Makefile
└── Report.pdf
```

Of course, you can add more files for your implementation(i.e. header files, additional C files). However, the project should be built by the command `make` and not allowed any additional command to build the project. And never change the app.c and debugfs file name(If you change it, the grading scripts can't work correctly).

Assignment #1: Process Tree Tracing

The purpose of this Assignment is tracing process from the leaf to `init` process and logging it using `debugfs`. By this Assignment, you can understand the `task_struct` which has information about the process and manage it. In Linux System, there is process tree to manage the processes in the system. The tree has `init` process, pid 1, as a root node and every process except `init` has one parent process.



In user space, process can get its pid using `getpid()` function and also its parent pid using `getppid()`. But, process can't know all of parent processes pid. In order to get all parent pid, you have to access kernel space. Every process in Linux System has `task_struct` which has the whole information for the process. In kernel space, you can manage `task_struct` to get the information about the process(i.e. pid, parent process `task_struct` pointer, etc.). You can access the parent process `task_struct` recursively and finally you can access `init` process.

You should follow the rules to build and *debugfs* file name. The build command is `make` using `Makefile`. The command `make` should include `insmod`. After `insmod` the module, you input the specific pid to */sys/kernel/debug/ptree/input*. Then, you can see the process tree branch from input pid to `init` process using command `cat` */sys/kernel/debug/ptree/ptree*. You can see the running processes using command `ps`.

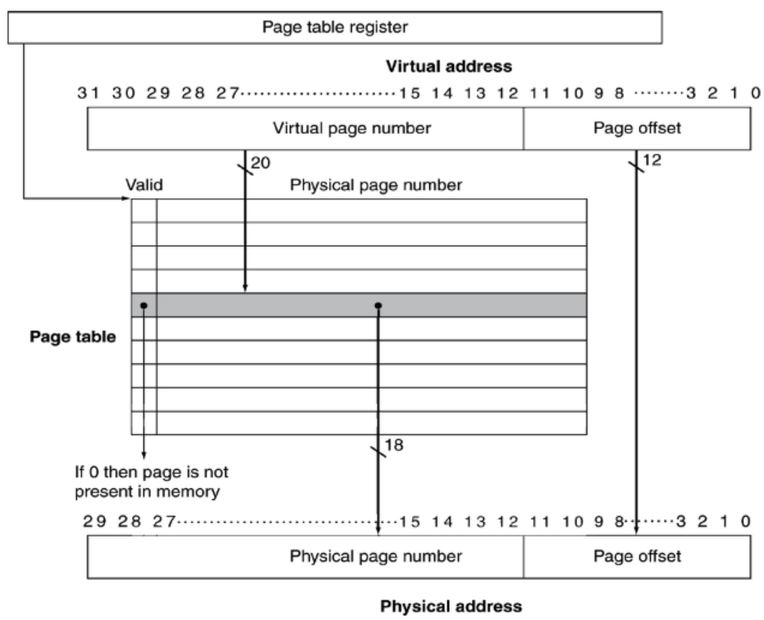
```
unix> ps
  PID TTY          TIME CMD
 2881 pts/0    00:00:00 sudo
 2882 pts/0    00:00:00 su
 2885 pts/0    00:00:00 bash
 2889 pts/0    00:00:00 ps
```

Then, you can choose one of them and write the pid to the input file. The whole stream of project running is like below:

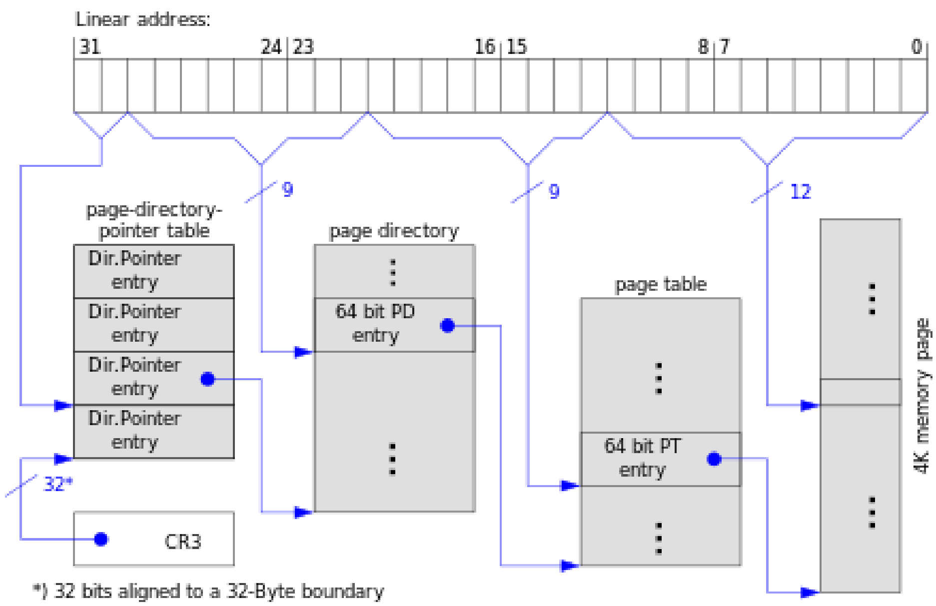
```
unix> make
        ...
unix> sudo su
unix> cd /sys/kernel/debug/ptree
unix> echo 2881 >> input
unix> cat ptree
init (1)
xfce4-panel (2306)
xfce4-terminal (2408)
bash (2413)
sudo (7066)
```

Assignment #2: Find Physical Address

Physical address is a memory address that is represented in the form of a binary number. Every process in Linux System has 4 GB memory space which is represented in the virtual address. But, this virtual address is not a actual address in physical memory. So, computer system has to translate the virtual address to physical address to access physical memory data. Every process gets the physical address from virtual address using Page Table which has the physical page number mapped to the virtual page number.



The Gentoo OS used in the Lab has 4-level page table. By using multi-level page table, the process has more compact page table area in memory than single page table system process.



`task_struct` in each process has `mm_struct` to manage memory area. `mm_struct` has the pointer of the top level page table entry(*pgd*). You can obtain the pointer of next level page table entry(*pud*) decoding *pgd* entry. Repeat this process, finally, you can obtain page table entry(*pte*) and find the physical address.

You should follow the rules to build and *debugfs* file name. The build command is `make` using `Makefile`. The command `make` should include `insmod` same as Assignment 1. After `insmod` the module, you run the application `app`, then you can see the result like this:

```
unix> sudo ./app
pid: 26136  vaddr: 220f010  paddr: 66c0c010
```

The *pid* is the process `app` pid and the *vaddr* is virtual address which allocated by `malloc`. As a result, the physical address is *paddr*.

Your source code should have the process of page walk through multi-level page tables. Don't use the other way to find physical address without page walk process. And you never change the file app.c and *debugfs* file name in the skeleton code.