# System Programming - Kernel Lab Report

2019 Spring, 2017-18570 Sungchan Yi

## 1. Kernel Lab

In kernel lab, we try a bit of kernel programming and our objective is to gain a deeper understanding of the Linux kernel.

## 2. Implementation

This lab consists of 2 parts: 1. Process Tree Tracing, 2. Finding Physical Address. For this task, we write a kernel module, using Linux kernel API.[1] Then we insert the module into the kernel and execute it. After we are done, we remove the module from the kernel, and remove any files created by the module.[2]

### 2.1 Part #1: Process Tree Tracing

For this task, we are given a process ID, then we trace the process tree up to the root process `init`. During the tracing, we use the `debugfs` API. Many of the functions necessary for this lab were given in the Lab description pdf, so I mainly looked at the functions written in the pdf.

First, create a directory with name `ptree`, and create files `input` and `ptree`. For the file `input`, we must only allow writes of PID to the file, so we set the `mode` to `S_IWUSR`. Set parent directory to `dir`, and pass `NULL` since there is no initial data. For the file `ptree`, only reads are allowed, so set the `mode` to `S_IRUSR`, and pass the `blob`. The `blob` will act as a buffer for writing to the file.

```
dir = debugfs_create_dir("ptree", NULL);
if (!dir) { // Error Checking
    printk("Cannot create ptree dir\n");
    return -1;
}
inputdir = debugfs_create_file("input", S_IWUSR, dir, NULL, &dbfs_fops);
ptreedir = debugfs_create_blob("ptree", S_IRUSR, dir, &blob);
```

With these settings, we have to print the process tree, starting from the `init` process. Therefore we need the pid, and the name of the parent process. This information is stored in `task_struct` of the current process, and this can be obtained by calling the function `pid_task`.

Furthermore, since we print the processes starting from `init`, it is natural to use a LIFO

[1] I had to refer to many documentations, and the linux github repo...

[2] Or you can reboot your machine, but not very efficient for debugging purposes...

structure, and the easiest one is to use recursion.[3] Thus if the PID is not 1, we call `print_pid` function recursively, and print to the buffer.

```
void print_pid(struct task_struct* task) {
    if(task -> pid > 1)
        print_pid(task -> real_parent); // if not init, recursive call
    size = snprintf(blob.data + blob.size, BUF - blob.size, "%s (%d)\n",
   task -> comm, task -> pid); // print to buffer
    blob.size += size;
}
```

The result could be check in the `/sys/kernel/debug/ptree` directory, by using[4]

```
# echo [process id] >> input
# cat ptree
```

### 2.2 Part #2: Find Physical Address

For this task, we try to convert a virtual address (VA) to physical address (PA). In the operating system level, a *page table* is used to map VA to PA. Since there are a lot of addresses, multi-level page table is used. Typically, a 4-level page table is used, but for this task, we assume the following environment, which uses a 5-level page table.

- OS: Ubuntu 18.04.2 LTS
- Kernel: 4.18.0-17-generic

When there are 5-levels of page tables, the search is performed in the following order,[5]

`pgd` (global) → `p4d`[6] → `pud` (upper) → `pmd` (middle) → `pte` (page table entry)

So we keep in mind that the search should be done in the above order. The initialization part was similar to Part 1, we omit the process here.
For `read_output` function, we get a `struct packet*` from `user_buffer`, which contains the PID. To get the `task_struct`, call `pid_task` as we did in Part 1. Then by using the virtual address (`vaddr`) in the `struct packet`, we iteratively obtain the offset for each level of the page table. The two things to note are that the `pgd` can be obtained from the `mm` field of the `task_struct`, and getting the `pte` will be done by calling the function `pte_offset_map`.

```
task = pid_task(find_get_pid(pckt -> pid), PIDTYPE_PID); // get task_struct

// get pgd from task -> mm
pgd = pgd_offset(task -> mm, pckt -> vaddr);
```

[3] Function call **stack**.
[4] #, to denote that this must be running on supervisor mode.
[5] *p* stands for *page*, *d* stands for *directory*.
[6] This level does not exists in the 4-level page table.

```
5
6  // get p4d from pgd
7  p4d = p4d_offset(pgd, pckt -> vaddr);
8
9  // get pud from p4d
10 pud = pud_offset(p4d, pckt -> vaddr);
11
12 // get pmd from pud
13 pmd = pmd_offset(pud, pckt -> vaddr);
14
15 // get pte from pmd
16 pte = pte_offset_map(pmd, pckt -> vaddr);
```

Then from `pte`, obtain the page by calling `pte_page` function, and set the physical address (`paddr`) of the packet to the physical address of the page by calling the function `page_to_phys`.

```
1  // get page
2  pg = pte_page(*pte);
3
4  // map page to physical address
5  pckt -> paddr = page_to_phys(pg);
```

Before running the module, set `PADDR` constant as `0x234512000` or `0x20000`.[7] To actually run the module, type in `./app` in the terminal to see `[TEST CASE] PASS`.

## 3. Conclusion

This is the first time doing kernel programming, and we can directly see the difference, compared to the usual programming we do. For instance, we can access the PID of the parent process, or the physical address. Although kernel programming was very new, it really teaches one a lot, and allows one to gain a better understanding of the system.

In my case, I first tried to program Part 2 by using a 4-level page table. For some reason, it worked, and now I guess that since the virtual address given to the process isn't that high enough, the entry in the `pgd` will be the 0-th entry, which is why I didn't have to do anything special for the first table.

[7] This value is specific to the environment you're using.