M1522.000800 System Programming
Fall 2018

System Programming Kernel Lab Report

Lee Young Jae

2016-11988

1. Kernel Lab

The goal of the kernel lab is to learn basic kernel module programming and understand the difference between kernel-level programming and user-level programming. To put the new feature into the kernel, we originally need to recompile the kernel. In my experience with installing Gentoo, it takes about 2 hours to compile, which is very annoying and risky. Instead, we can safely add new module with `debugfs` with very short time, since the modules we added disappear when we reboot the computer. The goal of this lab is adding two modules `ptree`, and `paddr`. What exactly each module does is introduced in the next chapter.

2. Implementation

The kernel developers have to follow the convention for Linux Kernel Module. A basic frame for Linux Kernel Module is `init_module` and `exit_module`. The formal is called when the kernel module is inserted to system, and the latter 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. Here is a basic structure for kernel module programming in `debugfs`:

```c
static ssize_t operation(struct file *fp,
                         const char __user *user_buffer,
                         size_t length,
                         loff_t *position)
{
    // Operation Details
}

static const struct file_operations dbfs_fops = {
    .operation = operation,
};

static int __init dbfs_module_init(void)
{
    // Some Codes
}

static void __exit dbfs_module_exit(void)
{
    // Some Codes
}
```

In each assignments, `ptree` and `paddr`, it has skeleton C code and build script. My task is implement to complete each codes. Fortunately, I did not have to fix the Makefile.

2.1. ptree

The purpose of this assignment is tracing process from the leaf to `init` process and logging it using `debugfs`. The followings are the steps of tracing process from the leaf to `init` in `write_pid_to_pid`. We initially store current task in the stack, then trace it by using `curr->real_parent`. The stored task information is popped into `stat` buffer. The printed information in `stat` is transfered to `struct debugfs_blob_wrapper *myblob`, which can hold as much data as we want.

1. Initialize `buffer`

```
for (i = 0; i < SIZE; i++) {
    stats[i] = '\0';
}
```

2. Get `pid` from `user_buffer`.

```
pid_t input_pid;
struct pid *pid;
sscanf(user_buffer, "%u", &input_pid);
pid = find_get_pid(input_pid);
```

3. Get `task_struct` from `pid` of 2.

```
curr = pid_task(pid, PIDTYPE_PID);
```

4. To print `pid` reversely, make a simple stack(or linked list)

```
struct stack {
    struct task_struct *task;
    struct stack *next;
};
struct stack *stack, *temp;
stack = (struct stack *) kmalloc(sizeof(struct stack), GFP_KERNEL);
temp = (struct stack *) kmalloc(sizeof(struct stack), GFP_KERNEL);
temp->next = NULL;
temp->task = curr;
```

5. Trace task struct until `pid` of the task is equal to 1. (In other words, the task is `init` struct.) The task information is stored in the stack.

```
while(1) {
    stack->next = temp;
    if (curr->pid == 1) break;
    curr = curr->real_parent;
    stack->task = curr;
    temp = stack;
    stack = (struct stack *) kmalloc(sizeof(struct stack), GFP_KERNEL);
}
stack = stack->next;
```

6. After it reaches to init, pop the stack value on the stat buffer.

```
while(1) {
    curr = stack->task;
    length += sprintf(stats + length, "%s␣(%d)\n", curr->comm, curr->pid);
    stack = stack->next;
    if (stack == NULL) break;
}
```

7. Now we complete write_pid_to_input. To use in init_module, we have to save this file operation in .write.

```
static const struct file_operations dbfs_fops = {
    .write = write_pid_to_input,
};
```

8. In __init dbfs_module_init function, define debugfs_blob_wrapper type. This struct would help us save data what we want, since strings are too big to store.

```
int stats_size;
int struct_size;
dir = debugfs_create_dir("ptree", NULL);
struct_size = sizeof(struct debugfs_blob_wrapper);
stats_size = 8192 * sizeof(char);
```

9. The debugfs_blob_wrapper type variable name is myblob. We initialize it with kmalloc since we cannot we malloc. Then, store the data and size as we get in the steps 1 to 6.

```
myblob = (struct debugfs_blob_wrapper *) kmalloc(struct_size, GFP_KERNEL);
myblob->data = (void *) stats;
myblob->size = (unsigned long) stats_size;
```

10. Now, create the file with permission 644. It finishes the initializing module.

```
inputdir = debugfs_create_file("input", 0644, dir, NULL, &dbfs_fops);
ptreedir = debugfs_create_blob("ptree", 0644, dir, myblob);
```

11. Finally, implement exit_module. Make free myblob to prevent memory leakage and remove created filesa and directory.

```
static void __exit dbfs_module_exit(void) {
    kfree(myblob);
    debugfs_remove(ptreedir);
    debugfs_remove(inputdir);
    debugfs_remove_recursive(dir);
}
```

2.2. paddr

The purpose of this assignment is finding pid, virtual memory address and physical memory address of app file. Virtual address is a memory address that points the next virtual or physical memory.

Using virtual address, a user can save/load data efficiently like cache. Thus, the virtual address is not a actual physical memory address and a computer system translates to communicate with virtual and physical memory. In each process, a process is allocated virtual memory address that has access to the actual physical memory address. 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. The task in this assignment is thus in fact get physical address from `pid` and virtual address. Finding them are exremely easy. The followings are how I implemented it.

1. Define data struct to read, then store data from `user_buffer`. We can syncronize `pckt` and `user_buffer` by pointing the same address. Now, we get correct `pid` and `vaddr`.

```
struct packet{
    pid_t pid;
    unsigned long vaddr;
    unsigned long paddr;
};
pckt = (struct packet *)user_buffer;
```

2. Then, get the current task as we implemented in `ptree`.

```
struct pid *pid;
pid = find_get_pid(pckt->pid);
task = pid_task(pid, PIDTYPE_PID);
```

3. Declare four page table offsets, and declare `page_addr` and `page_offset`.

```
pgd_t *pgd;
pud_t *pud;
pmd_t *pmd;
pte_t *pte;
unsigned long page_addr = 0;
unsigned long page_offset = 0;
```

4. Thanks to `p*d_offset` API, we can easily get each offsets from the previous offset or `mm_struct`.

```
pgd = pgd_offset(task->mm, pckt->vaddr);
pud = pud_offset(pgd, pckt->vaddr);
pmd = pmd_offset(pud, pckt->vaddr);
pte = pte_offset_kernel(pmd, pckt->vaddr);
```

5. Finally, get physical address. In this step, we use `PAGE_MASK` to get `page_addr` and `page_offset`. The physcial address is a combination of page address and page offset.

```
page_addr = pte_val(*pte) & PTE_PFN_MASK;
page_offset = pckt->vaddr & ~PTE_PFN_MASK;
pckt->paddr = page_addr | page_offset;
```

6. Create our output file, and don't forget to remove in exit phase.

```
static int __init dbfs_module_init(void) {
    dir = debugfs_create_dir("paddr", NULL);
    output = debugfs_create_file("output", 0644, dir, pckt, &dbfs_fops
    return 0;
}

static void __exit dbfs_module_exit(void) {
                debugfs_remove(output);
                debugfs_remove_recursive(dir);
}
```

3. Conclusion

In Kernel lab, I've learned how to implement kernel module programming. I've never used kernel information since I am just end user of the system programming. `kamlloc` function and `printk` function are meaningful to me. I did not understand why I have to use that instead of `malloc` and `printf`. In debugging process(not kernel debugging, my debugging), I use `dmesg` to show my logs. `printk` function was really helpful.

The most difficult problem was reboot problem. Unlike the previous project, I have to use Gentoo virtual machine since it is a kernel programming. After any error, I have to reboot to make the files again. If not, teminal says there is module running and kill my processor. Although Virtualbox is quite fast, reboot takes some time and makes me annoying.

The most surpsrising thing was that physical address is too big. `sudo ./app` says that my `pid`, `vaddr`, `paddr` value is `pid: 3401 vaddr: 12bc010 paddr:8000000070488010`. I think I might take some mistakes since the most significant bit in `paddr` is 1, and the following 7 bits are zero.

Bibliography

[1] http://jake.dothome.co.kr/pid (good reference about pid functions)

[2] https://stackoverflow.com/questions/14666991/how-to-use-the-debugfs-blob-wrapper-in-a-kernel-module (debugfs˙blob˙wrapper example)

[3] https://www.kernel.org/doc/htmldocs/filesystems/API-debugfs-create-file.html
https://www.kernel.org/doc/htmldocs/filesystems/API-debugfs-create-blob.html
(debugfs˙createfile, debugfs˙create˙blob)

[4] https://stackoverflow.com/questions/41090469/linux-kernel-how-to-get-physical-address-memory-management (vaddr to paddr)