

Prof. Heonyoung Yeom
System Programming (001)

Lab #4 / Kernel Lab

: Linux Module Programming

Report

Dept. of Computer Science and Engineering
2021-13194
Jaejun Ko

2023.5.21

Lab #4 Report

Kernel Lab : Linux Module Programming

Introduction

In this lab session, I will program two Linux modules, `ptree`, and `paddr`. By implementing these, I can understand *Linux Kernel Module Programming* based on *Debug File System* interface. `ptree` is tracing the process tree from a specific process id, and `paddr` is finding a physical address using a virtual address.

Specification

For Part 1, `ptree` should be implemented, and `paddr` is for Part 2.

Part 1. Process Tree Tracing(`ptree`)

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

In user space, a process can get its pid using the `getpid()` function and also its parent pid using `getppid()`. But, the process can't know all of the ancestor processes pid. To get all ancestor pid, access to kernel space is necessary. Every process in the Linux system has `task_struct` which has the whole information for the process. In kernel space, `task_struct` can be managed to get the information about the process(e.g. pid, parent process `task_struct` pointer, etc.). The access to the parent process `task_struct` can be conducted recursively until get into the `init` process.

The sequence of printing should be from the `init` to the leaf process in the tree.

Part 2. Find Physical Address(`paddr`)

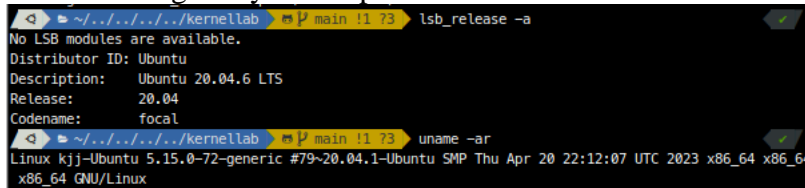
The physical address is a memory address that is represented in the form of a binary number. Historically, every process in a 32-bit Linux system has 4 GB of memory space which is represented in the virtual address. But, this virtual address is not an actual address in physical memory. So, the computer system has to translate the virtual address to a physical address to access physical memory data. Every process gets the physical address from the virtual address using a *page table* which has the physical page number mapped to the virtual page number. However, modern processors and Linux systems using 64-bit architecture support up to 256 TB of memory space. 48 bits are required to represent the virtual address. Therefore, 36 bits are used for the virtual page number in 64-bit Linux systems.

`task_struct` in each process has `mm_struct` to manage the memory area. `mm_struct` has the pointer of the top-level page table entry (*pgd*). You can obtain the pointer of the next level page table entry (*p4d*) decoding *pgd* entry. Repeat this process, finally, you can obtain the page table entry (*pte*) and find the physical address.

The module `paddr` should have the process of *page walk* through multi-level page tables.

Implementation

The following is my development environment.



```
~/.kernellab main !1 73 lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description:   Ubuntu 20.04.6 LTS
Release:      20.04
Codename:     focal
~/.kernellab main !1 73 uname -ar
Linux kjj-Ubuntu 5.15.0-72-generic #79-20.04.1-Ubuntu SMP Thu Apr 20 22:12:07 UTC 2023 x86_64 x86_64
x86_64 GNU/Linux
```

Part 1. Process Tree Tracing(ptree)

Conventionally, each module has a basic frame, which has two functions, which are called when inserted into the system and removed from the system, respectively. Also, it can create some files by using *debugfs*, a file system specially designed for debugging purposes available in the Linux kernel. The following is the init function and exit function, `dbfs_module_init` and `dbfs_module_exit`.

```
#define BUFFER_SIZE 100000
...

static struct debugfs blob wrapper blob;

static const struct file_operations dbfs_fops = {
    .write = write_pid_to_input,
};
...
static int __init dbfs_module_init(void) {
    // initialize blob
    static char buffer[BUFFER_SIZE];
    blob.data = buffer;

    // create ptree module directory
    dir = debugfs_create_dir("ptree", NULL);
    if (!dir) {
        printk("Cannot create ptree dir\n");
        return -1;
    }

    // create input file under ptree directory
    inputdir =
        debugfs_create_file("input", S_IRWXUGO, dir, blob.data, &dbfs_fops);
    if (!inputdir) {
        printk("Cannot create input dir\n");
        return -1;
    }
    // create output file under ptree directory
    ptreedir = debugfs_create_blob("ptree", S_IRWXUGO, dir, &blob);
    if (!ptreedir) {
        printk("Cannot create ptree dir\n");
        return -1;
    }

    printk("dbfs ptree module initialize done\n");
    return 0;
}

static void __exit dbfs_module_exit(void) {
    // Remove ptree module directory
    debugfs_remove_recursive(dir);
    printk("dbfs_ptree module exit\n");
}

module_init(dbfs_module_init);
module_exit(dbfs_module_exit);
```

In initialization, it initializes the `blob`, which will be used to contain some data for the output. After that, it creates a directory and files for the input(`input`) and the output(`ptree`). When the module is removed from the system, it removes created files and a directory recursively. Each file permits all access, although it indeed doesn't need to. For the input file,

it operates some file_operations, dbfs_fops. It executes write_pid_to_input if some data is written on the input file.

After the module has initialized, write_pid_to_input will be executed if some data is written on the input file. The implementation for write_pid_to_input is given below. By using copy_from_user, write_pid_to_input takes input and allocates it to the input_pid. From input_pid, the task can be found by using pid_task and find_get_pid. If the task is available, it starts to track the ancestors of the given task. After initializing a list task_list, add tasks updating curr to curr->real_parent while curr->pid == 1, or task is the init. For each task, I need to get a memory allocation on kernel virtual memory space using kmalloc. After that, it prints the ancestors to the output file from the init process to the given process. As I added tasks to the head of the list, it prints from the init to the leaf process. blob.size is added to check the size of the output. If it exceeds BUFFER_SIZE, it will make an error, but it will rarely happen.

```
typedef struct {
    struct task_struct *task;
    struct list_head list;
} task_node;

// write pid to input file
static ssize_t write_pid_to_input(struct file *fp,
                                const char *user_buffer, size_t length,
                                loff_t *position) {

    pid_t input_pid;
    // get pid from user_buffer
    if (copy_from_user(blob.data, user_buffer, length))
        return -EFAULT;
    sscanf(blob.data, "%u", &input_pid);

    // find task struct using pid
    curr = pid_task(find_get_pid(input_pid), PIDTYPE_PID);
    if (!curr)
        return -EINVAL;

    // print all ancestor processes using list
    LIST_HEAD(task_list);
    while(curr->pid != 1){
        task_node *curr_node = kmalloc(sizeof(task_node), GFP_KERNEL);
        curr_node->task = curr;
        list_add(&curr_node->list, &task_list);
        curr = curr->real_parent;
    }
    blob.size = snprintf(blob.data, BUFFER_SIZE,
                        "%s (%d)\n", curr->comm, curr->pid);

    task_node *p;
    list_for_each_entry(p, &task_list, list){
        blob.size += snprintf(blob.data + blob.size, BUFFER_SIZE - blob.size,
                            "%s (%d)\n", p->task->comm, p->task->pid);
    }
    return length;
}
```

Part 2. Find Physical Address(paddr)

In paddr, I only need to create a directory and an output file. Of course, it removes them when it exits. When the output is read, read_output is executed. It is named output since it is an output for the app, the tester.

```
static const struct file_operations dbfs_fops = {
    .read = read_output,
};

static int __init dbfs_module_init(void) {
    // create paddr module directory
    dir = debugfs_create_dir("paddr", NULL);
}
```

```

if (!dir) {
    printk("Cannot create paddr dir\n");
    return -1;
}

// create output file under module directory
output = debugfs_create_file("output", S_IRWXUGO, dir, NULL, &dbfs_fops);
if (!output) {
    printk("Cannot create output file\n");
    return -1;
}

printk("dbfs paddr module initialize done\n");
return 0;
}

static void __exit dbfs_module_exit(void) {
    // remove paddr module directory using debugfs_remove_recursive
    debugfs_remove_recursive(dir);
    printk("dbfs_paddr module exit\n");
}

module_init(dbfs_module_init);
module_exit(dbfs_module_exit);

```

The following function is `read_output`. When the output file is read, it is executed. Since the input is given as `packet_struct`, which has `pid`, `vaddr`, and `paddr`, I will use the struct directly. First, it copies the data of the output and allocates it to `pckt`, and gets the task again. Now, get `mm_struct` from the task, and use it to page walk. After getting the page table entry, get `pfn` from it. Since the physical page offset is equal to the virtual page offset, just copy it. By concatenating them, `paddr` was retrieved and saved to `pckt`. Finally, it copies the data of `pckt` to `user_buffer`, then all job is done.

```

struct packet {
    pid_t pid;
    unsigned long vaddr;
    unsigned long paddr;
};

static ssize_t read_output(struct file *fp, char __user *user_buffer,
                          size_t length, loff_t *position) {
    struct packet pckt;
    struct mm_struct *mm;
    pgd_t *pgd;
    p4d_t *p4d;
    pud_t *pud;
    pmd_t *pmd;
    pte_t *ptep, pte;
    unsigned long pfn, vpo;

    struct page *page = NULL;
    if (copy_from_user(&pckt, user_buffer, length))
        return -EFAULT;

    // find task_struct using pid
    task = pid_task(find_get_pid(pckt.pid), PIDTYPE_PID);
    if (!task)
        return -EINVAL;

    mm = task->mm;
    if (!mm)
        return -EINVAL;

    pgd = pgd_offset(mm, pckt.vaddr);
    if (pgd_none(*pgd) || pgd_bad(*pgd))
        return -EINVAL;

    p4d = p4d_offset(pgd, pckt.vaddr);
    if (p4d_none(*p4d) || p4d_bad(*p4d))
        return -EINVAL;

    pud = pud_offset(p4d, pckt.vaddr);

```

```

if (pud_none(*pud) || pud_bad(*pud))
    return -EINVAL;

pmd = pmd_offset(pud, pckt.vaddr);
if (pmd_none(*pmd) || pmd_bad(*pmd))
    return -EINVAL;

ptep = pte_offset_map(pmd, pckt.vaddr);
if (!ptep)
    return -EINVAL;

pte = *ptep;
pte_unmap(ptep);
if (!pte_present(pte))
    return -EINVAL;

page = pte_page(pte);
if (!page)
    return -EINVAL;

pfn = page_to_pfn(page);

vpo = pckt.vaddr & ~PAGE_MASK;
pckt.paddr = (pfn << PAGE_SHIFT) | vpo;

if (copy_to_user(user_buffer, &pckt, length))
    return -EFAULT;
return length;
}

```

Result

Part 1. Process Tree Tracing(ptree)

The following is the result of the execution. After making the module, it was inserted into the kernel. After checking which process is running by using `ps`, the pid of `bash` is written on the input. Then `write_pid_to_input` would be executed, and the results are written on `ptree`. By executing `cat ptree`, I could get the content of the output file. The result was correct. Finally, the module is removed from the kernel, and the make was cleaned.

```

root@kjj-Ubuntu:/home/kjj/Desktop/SystemProgramming-SNU/Lab/Lab4-KernelLab/kernelLab/ptree# make
make -C /lib/modules/5.15.0-72-generic/build M=/home/kjj/Desktop/SystemProgramming-SNU/Lab/Lab4-KernelLab/kernelLab/ptree modules;
make[1]: 디렉터리 '/usr/src/linux-headers-5.15.0-72-generic' 들어감
CC [M] /home/kjj/Desktop/SystemProgramming-SNU/Lab/Lab4-KernelLab/kernelLab/ptree/dbfs_ptree.o
In file included from ./include/linux/wait.h:7,
                 from ./include/linux/wait_bit.h:8,
                 from ./include/linux/fs.h:6,
                 from ./include/linux/debugfs.h:15,
                 from /home/kjj/Desktop/SystemProgramming-SNU/Lab/Lab4-KernelLab/kernelLab/ptree/dbfs_ptree.c:1:
./include/linux/list.h:24:2: warning: ISO C90 forbids mixed declarations and code [-Wdeclaration-after-statement]
 24 | struct list_head name = LIST_HEAD_INIT(name)
    | ^~~~~
/home/kjj/Desktop/SystemProgramming-SNU/Lab/Lab4-KernelLab/kernelLab/ptree/dbfs_ptree.c:36:3: note: in expansion of macro 'LIST_HEAD'
 36 |     LIST_HEAD(task_list);
    |     ^~~~~
/home/kjj/Desktop/SystemProgramming-SNU/Lab/Lab4-KernelLab/kernelLab/ptree/dbfs_ptree.c:45:3: warning: ISO C90 forbids mixed declarations and code [-Wdeclaration-after-statement]
 45 |     task_node *p;
    |     ^~~~~
MODPOST /home/kjj/Desktop/SystemProgramming-SNU/Lab/Lab4-KernelLab/kernelLab/ptree/Module.symvers
CC [M] /home/kjj/Desktop/SystemProgramming-SNU/Lab/Lab4-KernelLab/kernelLab/ptree/dbfs_ptree.mod.o
LD [M] /home/kjj/Desktop/SystemProgramming-SNU/Lab/Lab4-KernelLab/kernelLab/ptree/dbfs_ptree.ko
BTF [M] /home/kjj/Desktop/SystemProgramming-SNU/Lab/Lab4-KernelLab/kernelLab/ptree/dbfs_ptree.ko
Skipping BTF generation for /home/kjj/Desktop/SystemProgramming-SNU/Lab/Lab4-KernelLab/kernelLab/ptree/dbfs_ptree.ko due to unavailability of vmlinux
make[1]: 디렉터리 '/usr/src/linux-headers-5.15.0-72-generic' 나감
sudo insmod dbfs_ptree.ko
root@kjj-Ubuntu:/home/kjj/Desktop/SystemProgramming-SNU/Lab/Lab4-KernelLab/kernelLab/ptree# cd /sys/kernel/debug/ptree
root@kjj-Ubuntu:/sys/kernel/debug/ptree# ps
  PID TTY          TIME CMD
  9866 pts/2    00:00:00 sudo
  9867 pts/2    00:00:00 su
  9868 pts/2    00:00:00 bash
 10175 pts/2    00:00:00 ps
root@kjj-Ubuntu:/sys/kernel/debug/ptree# echo 9868 >> input
root@kjj-Ubuntu:/sys/kernel/debug/ptree# cat ptree
systemd (1)
systemd (1948)
gnome-shell (2176)
terminator (2965)
zsh (9683)
sudo (9866)
su (9867)
bash (9868)
root@kjj-Ubuntu:/sys/kernel/debug/ptree# cd /home/kjj/Desktop/SystemProgramming-SNU/Lab/Lab4-KernelLab/kernelLab/ptree
root@kjj-Ubuntu:/home/kjj/Desktop/SystemProgramming-SNU/Lab/Lab4-KernelLab/kernelLab/ptree# make clean
make -C /lib/modules/5.15.0-72-generic/build M=/home/kjj/Desktop/SystemProgramming-SNU/Lab/Lab4-KernelLab/kernelLab/ptree clean;
make[1]: 디렉터리 '/usr/src/linux-headers-5.15.0-72-generic' 들어감
CLEAN /home/kjj/Desktop/SystemProgramming-SNU/Lab/Lab4-KernelLab/kernelLab/ptree/Module.symvers
make[1]: 디렉터리 '/usr/src/linux-headers-5.15.0-72-generic' 나감
sudo rmmod dbfs_ptree.ko

```

Part 2. Find Physical Address(paddr)

The following is the result of the execution. After making the module and app, it was inserted into the kernel. By executing app, I could check the module works very well. After that, I removed it from the kernel and clean the make again.

```
root@kjj-Ubuntu:/home/kjj/Desktop/SystemProgramming-SNU/Lab/Lab4-Kernellab/kernellab/paddr# make
make -C /lib/modules/5.15.0-72-generic/build M=/home/kjj/Desktop/SystemProgramming-SNU/Lab/Lab4-Kernellab/kernellab/paddr modules;
make[1]: 디렉터리 '/usr/src/linux-headers-5.15.0-72-generic' 들어감
CC [M] /home/kjj/Desktop/SystemProgramming-SNU/Lab/Lab4-Kernellab/kernellab/paddr/dbfs_paddr.o
MODPOST /home/kjj/Desktop/SystemProgramming-SNU/Lab/Lab4-Kernellab/kernellab/paddr/Module.symvers
CC [M] /home/kjj/Desktop/SystemProgramming-SNU/Lab/Lab4-Kernellab/kernellab/paddr/dbfs_paddr.mod.o
LD [M] /home/kjj/Desktop/SystemProgramming-SNU/Lab/Lab4-Kernellab/kernellab/paddr/dbfs_paddr.ko
BTF [M] /home/kjj/Desktop/SystemProgramming-SNU/Lab/Lab4-Kernellab/kernellab/paddr/dbfs_paddr.ko
Skipping BTF generation for /home/kjj/Desktop/SystemProgramming-SNU/Lab/Lab4-Kernellab/kernellab/paddr/dbfs_paddr.ko due to unavailability of vmlinux
make[1]: 디렉터리 '/usr/src/linux-headers-5.15.0-72-generic' 나감
gcc -o app app.c;
sudo insmod dbfs_paddr.ko
root@kjj-Ubuntu:/home/kjj/Desktop/SystemProgramming-SNU/Lab/Lab4-Kernellab/kernellab/paddr# ./app
[TEST CASE] PASS
root@kjj-Ubuntu:/home/kjj/Desktop/SystemProgramming-SNU/Lab/Lab4-Kernellab/kernellab/paddr# make clean
make -C /lib/modules/5.15.0-72-generic/build M=/home/kjj/Desktop/SystemProgramming-SNU/Lab/Lab4-Kernellab/kernellab/paddr clean;
make[1]: 디렉터리 '/usr/src/linux-headers-5.15.0-72-generic' 들어감
CLEAN /home/kjj/Desktop/SystemProgramming-SNU/Lab/Lab4-Kernellab/kernellab/paddr/Module.symvers
make[1]: 디렉터리 '/usr/src/linux-headers-5.15.0-72-generic' 나감
rm app;
sudo rmmod dbfs_paddr.ko
```

Discussion

What was Difficult?

The most difficult part when implementing the kernel is the fact that I need to reboot if the module makes some error. To prevent this, I need to check all values of system calls or functions. Furthermore, I could understand why TAs recommended using a Virtual Machine.

Also, finding the macros and functions was so difficult. In the skeleton code, it included `<asm/pgtable.h>` in `dbfs_paddr.c`. However, most of the function I wanted to use was implemented in `<linux/pgtable.h>`, which made me confused.

Something New and Surprising

Before this lab session, I thought that we cannot access kernel virtual memory space or can access it only in some restricted ways. However, accessing to kernel VM is relatively easy by copying from or to the kernel VM. Furthermore, The fact that dynamic memory allocation in kernel VM is available was surprising.

Conclusion

In this lab session, I could understand *Linux Kernel Module Programming* based on *Debug File System* interface. I implemented two Linux modules, `ptree` and `paddr`, which traces the process tree from a specific process id and finds a physical address using a virtual address, respectively. Furthermore, by exploiting kernel virtual memory space, I could dynamically allocate some memory and use a list. The kernel module programming was difficult since we need reboot for every kernel panic, caused by error on my custom module.