System Programming

Kernel lab: Linux Module Programming

2017-18538 황선영

1. Goal of kernel lab

이번 lab에서는 Debug File System interface에 기반한 Linux Kernel Module Programming을 이해하는 것이 목표이다. 이를 이용하여 특정 process id를 tracing하여 조상을 추적하거나, virtual address를 이용하여 physical address를 찾는다.

2. Implementation

① dbfs_ptree.c

```
static ssize_t write_pid_to_input(struct file *fp,
                                  const char __user *user_buffer,
                                  size_t length,
                                  loff_t *position)
{
        pid_t input_pid;
        char temp_str[LENGTH_MAX];

        //initialize to 0
        memset(out_str, 0, LENGTH_MAX);

        sscanf(user_buffer, "%u", &input_pid);
        if(copy_from_user(temp_str, user_buffer, length))
                return -EFAULT;

        //find task_struct using input_pid
        curr = pid_task(find_get_pid(input_pid), PIDTYPE_PID);
        //check
        if(!curr) return -EINVAL;

        // Tracing process tree from input_pid to init(1) process
        while(curr->pid != 0)
        {
                // Make Output Format string: process_command (process_id)
                length_temp = snprintf(temp_str, LENGTH_MAX, "%s (%d)\n", curr->
comm, curr->pid);
                //append the last result
                snprintf(temp_str+length_temp, LENGTH_MAX-length_temp, out_str);
                length_total += length_temp;
                strcpy(out_str, temp_str);
                curr = curr -> parent;
        }
        return length;
}
```

write_pid_to_input 함수는 kernel space에 input된 pid를 쓰는 함수로, write file operation인 dbfs_fops_write가 call 되었을 때 실행된다. 결과값은 out_str에 저장되어 이후 read할 때 사용된다. user_buffer을 parameter로 받아 sscanf 함수를 이용하여 그 안의 값을 input_pid에 넣는다. 이는 커맨드로 받은 pid이다. 이후 user_buffer를 temp_str를 복사한다. 그 후 curr이라는 task_struct

에 현재 입력받은 pid의 task_struct를 넣는다. 그 후 현재 pid부터 맨 처음 조상까지의 process name과 pid를 recursively하게 받아 out_str에 저장한다. 이때 length_total은 out_str의 length이다.

```
static ssize_t read_output(struct file *fp,
                                char __user *user_buffer,
                                size_t length,
                                loff_t *position)
{
        //copy to user_buffer
        return simple_read_from_buffer(user_buffer, length, position, out_str, l
ength_total);
}

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

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

입력받은 pid로부터 정보(프로세스의 이름과 id)를 write함수를 이용하여 알아냈으므로 그 결과를 read_output함수를 이용하여 읽어낸다. 이때 simple_read_from_buffer 함수를 이용하여 out_str의 값을 user_buffer로 읽어온다.

두 file_operation은 user와 kernel의 연결고리이다. User space와 kernel space는 서로 접근하지 못하는 영역이기 때문에 데이터를 전달하는 함수를 file_operation과 연결하여 원하는 기능을 구현한다. dbfs_fops_write에는 write_pid_to_input함수를 연결하고, dbfs_fops_read에는 read_output을 연결했다

```
static int __init dbfs_module_init(void)
{
        // Implement init module code

        dir = debugfs_create_dir("ptree", NULL);

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

        inputdir = debugfs_create_file("input", 00700, dir, NULL, &dbfs_fops_wri
te);
        if(!inputdir)
        {
                printk("Cannot create inputdir file\n");
                return -1;
        }

        ptreedir = debugfs_create_file("ptree", 00700, dir, NULL, &dbfs_fops_rea
d); // Find suitable debugfs API
        if(!ptreedir)
        {
                printk("Cannot create ptreedir file\n");
                return -1;
        }


        printk("dbfs_ptree module initialize done\n");
```

```
        return 0;
}


static void __exit dbfs_module_exit(void)
{
        // Implement exit module code

        //remove recursively a directory tree in debugfs that was previously
        //created with a call to another debugfs function
        debugfs_remove_recursive(dir);
        printk("dbfs_ptree module exit\n");
}

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

dbfs_module_init 함수는 kernel module이 시스템에 삽입되었을 때 call되는 함수이고, dbfs_module_exit 함수는 kernel module이 시스템에서 삭제될 때 call되는 함수이다. dbfs_module_init 함수에서는 debugfs API를 이용하여 디렉토리와 파일을 만든다. ptree 디렉토리 안에는 input과 ptree 파일이 있는데, input은 입력을 받는 곳이고 ptree는 process tree를 만드는 곳이다. dbfs_module_exit 함수에서는 debugfs_remove_recursive 함수를 이용하여 생성되었던 tree를 없앤다.

② dbfs_paddr.c

Virtual address로부터 page table walk를 이용해 physical address를 찾는 프로그램을 구현하였다.

```
long power(long base, unsigned int exp)
{
        long result = 1;
        while(exp!=0)
        {
                result *= base;
                --exp;
        }
        return result;
}
```

power 함수는 base의 exp 제곱수를 return하는 함수이다. read_output 함수에서 쓰인다.

```
static ssize_t read_output(struct file *fp,
                        char __user *user_buffer,
                        size_t length,
                        loff_t *position)
{
        pid_t input_pid;
        long int vaddr = 0;
        int i;
        unsigned char kernel_buffer[LENGTH_MAX];
        unsigned char out_buf[LENGTH_MAX];
        int length_output = MIN(length, LENGTH_MAX);

        struct mm_struct *mm;
        pgd_t *pgdp;
        p4d_t *p4dp;
        pud_t *pudp;
        pmd_t *pmdp;
        pte_t *ptep;
        phys_addr_t pfn;
```

여러 변수와 page table walk를 위한 메모리 관련 변수를 선언하였다.

```
        memset(kernel_buffer, 0, LENGTH_MAX);
        memset(out_buf, 0, LENGTH_MAX);

        if(copy_from_user(kernel_buffer, user_buffer, length_output))
                return -EFAULT;

        //get pid
        input_pid = (int)(kernel_buffer[1])*16*16+(int)(kernel_buffer[0]);
        //get the vaddr input(48bit)
        for(i=0;i<6;i++) vaddr += (long)(kernel_buffer[8+i])*power(16*16,i);

        //page table walk(procedure: pgd-p4d=pud-pmd-pte)
        //get task_struct
        task = pid_task(find_get_pid(input_pid), PIDTYPE_PID);
        if(!task) return -EINVAL;
        mm = task->mm; //get mm_struct
        pgdp = pgd_offset(mm, vaddr); //pgd pointer
        if(pgd_none(*pgdp) || pgd_bad(*pgdp)) return -EINVAL; //check invalid
        p4dp = p4d_offset(pgdp, vaddr); //p4d pointer
        if(p4d_none(*p4dp) || p4d_bad(*p4dp)) return -EINVAL; //check invalid
        pudp = pud_offset(p4dp, vaddr); //pud pointer
        if(pud_none(*pudp) || pud_bad(*pudp)) return -EINVAL; //check invalid
        pmdp = pmd_offset(pudp, vaddr); //pmd pointer
        if(pmd_none(*pmdp) || pmd_bad(*pmdp)) return -EINVAL; //check invalid
        ptep = pte_offset_kernel(pmdp, vaddr); //pte pointer
        if(pte_none(*ptep) || !pte_present(*ptep)) return -EINVAL; //check inval
id
        pfn = pte_pfn(*ptep); //page frame number
```

먼저 physical address를 찾기 위해 virtual address가 필요하므로 pid와 vaddr 변수에 process id와 virtual address를 받는다.

이후 page table walk 과정을 거쳐 page frame number를 얻는다. 이때 리눅스 헤더 파일에 있는 매크로인 xxx_offset, xxx_none, xxx_bad 등을 이용하여 각각의 포인터를 받고 invalid를 체크한다.

```
        for(i=0;i<length_output;i++)
        {
                if(i<16) out_buf[i] = kernel_buffer[i];
                else if(i<17) out_buf[i] = kernel_buffer[i-8];
                else if(i<18)
                {
                        out_buf[i] = (kernel_buffer[i-8])%16;
                        out_buf[i] += (pfn%16)*16;
                        pfn /= 16;
                }
                else
                {
                        out_buf[i] = (pfn%16);
                        pfn /= 16;
                        out_buf[i] += (pfn%16)*16;
                        pfn /= 16;
                }
        }
        out_buf[length_output] = NULL;

        return simple_read_from_buffer(user_buffer, length, position, out_buf, l
ength_output);
}

static const struct file_operations dbfs_fops = {
        // Mapping file operations with your functions
        .read = read_output,
};
```

결과를 얻는 out_buf에 넣는 과정이다. 15번째 요소까지는 그대로 유지하고, 16번째 요소는 vaddr의 lower 8bit만 넣는다. 17번째 요소는 vaddr의 lower 12~9 bits만 유지하고 paddr의 lower 4 bits를 넣는다. 그리고 나머지 요소를 세팅하고 out_buf의 마지막 요소에는 NULL을 넣는다.

file_operation은 user와 kernel의 연결고리이므로 file_operations와 함수를 매핑하여 kernel에서 원하는 동작을 하도록 한다. kernel에서 physical address를 읽어와야 하므로 read_output 함수를 mapping했다.

```
static int __init dbfs_module_init(void)
{
        // Implement init module
        dir = debugfs_create_dir("paddr", NULL);

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

        // Fill in the arguments below
        output = debugfs_create_file("output", 00700, 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)
{
        // Implement exit module
        debugfs_remove_recursive(dir);
        printk("dbfs_paddr module exit\n");
}

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

dbfs_module_init 함수는 kernel module이 시스템에 삽입되었을 때 call되는 함수이고, dbfs_module_exit 함수는 kernel module이 시스템에서 삭제될 때 call되는 함수이다. dbfs_module_init 함수에서는 debugfs API를 이용하여 디렉토리와 파일을 만든다. 만든 디렉토리는 paddr이고 만든 파일은 output이다. dbfs_module_exit 함수에서는 debugfs_remove_recursive 함수를 이용하여 생성되었던 디렉토리를 없앤다.

```
root@hwang-VirtualBox:/home/hwang/Desktop/kernellab-handout/paddr# ./app
[TEST CASE]     PASS
```

Test case를 통과한 모습이다.

3. Conclusion

Kernel space를 user space에서 다루는 방법을 배우게 되었다. Debugfs API와 file operation의 개념과 구현이 조금 까다로웠지만 user가 kernel space를 다룰 수 있다는 점이 신선하게 다가왔다.

dbfs_paddr.c를 구현할 때는 리눅스 헤더파일에 있는 매크로를 이용하여 page table walk를 수행할 때 가장 이해가 어려웠지만 그 동작 원리에 놀라웠다. 그리고 output buffer에 physical address를 mapping할 때 숫자 계산이 까다로웠다. 이 숫자 계산을 하면서 어떻게 virtual address가 physical address로 translate되는지 배울 수 있었다.