Kernel Lab

2017-19428

컴퓨터공학과 서준원

1. Objective

이 랩의 목표는 커널 프로그래밍을 체험해 보는 것이었다. 커널 프로그래밍을 하는 방법과 유저 스페이스에서 커널을 다루는 방법을 경험해 볼 수 있었다. 이를 위해 다양한 개념들을 익힐 수 있었다.

본 과제에서 사용한 것들은 아래와 같다.

- Debugfs Api (Debug File System)

  이는 User Space에서 커널을 다룰 수 있도록 해주는 Api들로 이루어져 있었다. 이를 구체적으로 사용한 예시는 본문에 작성하였다.

- Kernel Module

  Linux OS에서 운영체제의 일부인 커널은 작은 단위인 모듈들로 쪼개져있다. 따라서 내가 커널 영역에서 특정 작업을 실행하고 싶을 때 커널 모듈을 추가하거나 제거할 수 있었다. Insmod란 명령어로 Load를 하고, Lsmod란 명령어로 커널이 로드되었는지 체크하며, rmmod로 모듈을 제거할 수 있었다.

2. Implementation

위 과제는 여러 리눅스 커널 Api를 이용하여 모듈을 만들고, 이를 실제로 Load 하여 실행하는 것이었다. 첫 번째 과제는 프로세스의 ID를 인풋으로 받고 부모 프로세스들의 ID를 출력하는 것이었다. 두 번째 과제는 app의 PID와 virtual address로 Physica Address를 알아내는 것이었다. 이는 페이지 테이블에 접근하는 것이었다.

코드 중간 중간에 상세 설명을 첨부하였다.

A. Process Tree Tracing

이는 커널 모듈의 헤더 파일이다. License는 적당한 권한을 주는 부분이다.

```
3. #include <linux/debugfs.h>
4. #include <linux/kernel.h>
5. #include <linux/module.h>
6. #include <linux/uaccess.h>
7. #define Buffer_Size 200000
8. MODULE_LICENSE("GPL");
```

이 함수는 **Process** 의 **Pid** 를 출력하는 함수이다. **task struct** 구조체는 프로세스의 다양한 정보를 갖고 있다. 그 중에서도 **real parent** 는 그 프로세스의 부모의 **task struct** 를 가르킨다. 또한 **comm** 는 프로세스의 이름을 가리키기 때문에 이를 사용했다. 모든 프로세스의 **root** 인 **pid** 가 **1** 인 프로세스 **(init)** 까지 올라가기 위해 재귀함수를 구현하였다. 그리고 재귀가 종료되면 루트부터 출력 버퍼에 포맷 스트링을 담아주었다. 이를 유저 스페이스에 출력하기 위해 **Debugfs** 의 **wrapper** 를 사용하였다.

```
void printpid(struct task_struct *task)
{
        if(task->pid != 1)
        {
                printpid(task->real_parent);
        }
        wrapper.size += snprintf(wrapper.data + wrapper.size, Buffer_Size -
wrapper.size , "%s (%d)\n", task->comm, task->pid);
}

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;

       sscanf(user_buffer, "%u", &input_pid);
       curr = pid_task(find_get_pid(input_pid), PIDTYPE_PID);
                  // Find task_struct using input_pid. Hint: pid_task

       // Tracing process tree from input_pid to init(1) process

                  wrapper.size = 0;
                  printpid(curr);
       // Make Output Format string: process_command (process_id)

       return length;
}
```

이것은 **file operation structure** 로 개발환경에서 **file operation** 을 하기 위한 인터페이스를 조정하는 부분이다. **File write** 를 하게 되면 위에 함수가 실행되도록 설정하였다.

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

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


        dir = debugfs_create_dir("ptree", NULL);
```

Input 과 **ptree** 파일을 생성하기 위해 **debugfs_create_dir** 을 사용했다. 기본 경로에 디렉토리를 생성하기 위해 **parameter** 로 **NULL** 을 설정하였다.

```
        if (!dir) {
                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, &wrapper);
                  static char buffer[Buffer_Size];
                  wrapper.data = buffer;
                  // Find suitable debugfs API
```

Input 의 경우 **pid** 를 쓰는 것만 가능해야 하므로 모드에 **S_IWUSR** 을 설정하였고, **dbfs_fops** 인 파일 구조체를 넣어주었다. **Blob** 을 만드는 과정에서는 아웃풋 파일을 작성할 버퍼를 만들어서 이를 넣어주었다. 버퍼는 **static** 으로 생성해주었다.

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

        return 0;
}

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

        debugfs_remove_recursive(dir);
```

모듈을 제거할 때 만든 디렉토리를 삭제해준다.

```
        printk("dbfs_ptree module exit\n");
}
```

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

B. Find Physical Address

위와 다른 내용인 코드만 첨부하였다. 나머지 커널 모듈을 삽입하거나 제거하기 위한 부분, 그리고 인풋을 받는 부분은 거의 일치한다. 여기서는 virtual address를 가지고 physical address를 찾는 함수를 첨부했다.

리눅스에서는 페이지 테이블을 multi-level page table을 사용한다. 내 경우는 리눅스의 버전이 4.18이기 때문에 app.c에서 초기 주소 값을 바꿔주었다. App.c로부터 packet이란 구조체를 넘겨받기 때문에 동일한 구조체를 정의하여 사용하였다. 나의 리눅스 환경은 x86-64이다. 따라서 총 48비트의 virtual address를 physical address로 전환하는 과정을 거쳤다. 4KB page table을 사용하기 때문에 하위 12비트를 page offset으로 사용하였다.

```
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 *temp = (struct packet*) user_buffer;
        pgd_t *pgd;
        pud_t *pud;
        pmd_t *pmd;
        pte_t *pte;

        unsigned long vpn[4];
        unsigned long vpo;
```

bit mask 를 사용하여 각각 level 의 page table 마다의 offset 을 계산했다.

```
        vpn[0] = ((temp->vaddr) >> 39 ) & flag;
        vpn[1] = ((temp->vaddr) >> 30 ) & flag;
        vpn[2] = ((temp->vaddr) >> 21 ) & flag;
        vpn[3] = ((temp->vaddr) >> 12 ) & flag;
```

```
        vpo = (temp->vaddr) & 0xFFF;

        task = pid_task(find_get_pid(temp->pid), PIDTYPE_PID);

        pgd = task->mm->pgd;
        pud = (pud_t *)(((pgd+vpn[0]) -> pgd & 0xFFFFFFFFF000) +
PAGE_OFFSET);
        pmd = (pmd_t *)(((pud+vpn[1]) -> pud & 0xFFFFFFFFF000) +
PAGE_OFFSET);
        pte = (pte_t *)(((pmd+vpn[2]) -> pmd & 0xFFFFFFFFF000) +
PAGE_OFFSET);
        temp->paddr = (((pte+vpn[3]) -> pte & 0xFFFFFFFFF000) + vpo);
```

페이지 테이블에서 0 번부터 address 를 mapping 하는 것이 아니기 때문에 page_offset 만큼 전환해주었다.

```
        return length;

       // Implement read file operation
}
```

3. Conclusion

커널을 직접 다룬다는 점에서 매우 흥미로웠다. 하지만 낮은 레벨의 프로그래밍인 만큼 레퍼런스 할 자료가 직접적으로 와 닿지 않아서 어려움을 많이 겪었다. 특히 debugfs Api나 kernel module을 다루기 위한 task_struct 등에 대한 자료를 찾아보는 데에 많은 시간이 필요했다.

하지만 이 과정에서 다양한 Api를 찾아보는 방법을 알 수 있었다. 또한 사용자 레벨과 커널 레벨의 프로그래밍이 어떻게 차이가 나는 지를 알 수 있었으며, User Space에서 Kernel 영역을 다루기 위해서는 어떻게 해야 하는 지 배울 수 있었다.