시스템 프로그래밍 kernel lab 보고서

2017-12751 컴퓨터공학부 이동학

이번 lab에서는 process id로부터 process tree를 tracing 하고 virtual address를 이용하여 physical address를 찾는 코드를 구현하면서 debug file system interface를 이용한 linux kernel module programming을 이해하는 것을 목표로 하였습니다

1. ptree.c

```
void printpid(struct task_struct *task)
{
        if(task->pid != 1)
        {
                printpid(task->real_parent);
        }
        wrapper.size += snprintf(wrapper.data + wrapper.size, 200000 - 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;
}
```

process의 pid를 출력하는 함수로, real_parent와 comm을 이용해 pid가 1인 프로세스까지 올라가는 재귀함수를 구현하였습니다. 재귀가 끝나면 역으로 추적하면서 출력 버퍼에 포맷 스트링을 담아주었고, 이를 출력하기 위해 wrapper를 사용하였습니다.

```c
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);

        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[200000];
        wrapper.data = buffer;
        // Find suitable debugfs API


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

        return 0;
}

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

        debugfs_remove_recursive(dir);
        printk("dbfs_ptree module exit\n");
}

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

file write 를 하면 write_pid_to_input 을 실행하게 하고 input 과 ptree 파일 생성을 위해 debugfs_create_dir 을 사용하였습니다. input 의 경우 pid 만 써야하므로 S_IWUSR 모드를 사용하고 dbfs_fops 구조체를 넣어주었습니다. blob 의 경우 output 파일을 작성할 버퍼를 만들어 이를 넣어주었습니다. 마지막으로 모듈을 제거할 때 만든 디렉토리를 삭제합니다.

2. paddr.c

```
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;

    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);

    return length;

        // Implement read file operation
}
```

mmap fail 이 발생해서 paddr 을 0x252345000 으로 바꿔주었고, app.c 에서 packet 구조체를 넘겨받으므로 같은 구조체를 정의하여 사용하였습니다. x86-64 환경이므로 48bit 의 virtual address 를 physical address 로 전환하였고, 4kb page table 을 사용하므로 하위 12bit 를 page offset 으로 사용하였습니다. 위 함수에서는 bit mask 를 사용하여 각 level 의 page table 마다 offset 을 계산해주고 mapping 을 할 때 page offset 만큼 전환해주었습니다.

후기: 커널을 직접 다루는 것은 신기하고 재미있는 경험이었지만, low level programming인 만큼 참고할 자료가 많지 않고 debug file system interface나 kernel module의 task_struct등이 직관적으로 다가오지 않아서 많이 어려웠습니다. 하지만 그 덕분에 kernel level programming과 user level programming의 차이를 배우고 kernel data structure의 구조와 메커니즘을 익힐 수 있었습니다.