

# 운영체제 실습 Report

실습 제목: Assignment 2

실습일자: 2022년 09월 15일 (목)

제출일자: 2022년 10월 13일 (목)

학 과: 컴퓨터정보공학부

담당교수: 최상호 교수님

실습분반: 금요일 5,6교시

학 번: 2018202065

성 명: 박 철 준

● Introduction

1. 제목

Assignment 2

2. 과제 요구사항

- ftracehooking.h : ftracehooking.c 및 ioftacehooking에서 사용하는 header를 정의한다.
- ftracehooking.c :
  1. ftrace 시스템콜을 hooking하여 ftrace 함수로 대체하는 커널 모듈 코드를 구현하라.
  2. pid\_t sys\_fttrace(pid\_t pid);를 사용하라.
  3. static asmlinkage int ftrace(const struct pt\_regs \*regs) 사용하라
  4. ioftacehooking.c와 연동해야 하며 EXPORT\_SYMBOL() 사용해야 한다.
- ioftacehooking.c :
  1. open, read, write, lseek, close 시스템콜을 hooking하여 ftrace\_open, ftrace\_read, ftrace\_write, ftrace\_lseek, ftrace\_close함수로 대체하는 커널 모듈 코드를 구현하라.
  2. static asmlinkage long ftrace\_open(const struct pt\_regs \*regs) 사용해야 한다.
- Makefile :
  - 1.ftracehooking.ko 파일과 ioftacehooking.ko 파일이 동시에 생성되도록 작성한 Makefile을 구현하라.

- Conclusion & Analysis

- ftracehooking.h의 구현 코드

```
1 #include <linux/module.h>
2 #include <linux/highmem.h>
3 #include <linux/kallsyms.h>
4 #include <linux/syscalls.h>
5 #include <asm/syscall_wrapper.h>
6 #include <linux/unistd.h>
7 #include <linux/string.h>
```

ftracehooking.c, ioftracehooking.c에서 사용하는 헤더파일들에 대한 정의이다.

- ftracehooking.c의 구현 코드

```
1 #include "ftracehooking.h"
2
3 #define __NR_ftrace 336
4
5 int open_count=0; // open 함수 카운트 변수
6 int read_count=0; // read 함수 카운트 변수
7 int close_count=0; // close 함수 카운트 변수
8 int lseek_count=0; // lseek 함수 카운트 변수
9 int write_count=0; // write 함수 카운트 변수
10 int read_byte=0; // read 바이트 카운트 변수
11 int write_byte=0; // write 바이트 카운트 변수
12 char kernel_buffer[1000] = {0,}; // 커널 버퍼 변수
13 pid_t given_pid=0; // 프로세스 피아이드를 저장할 변수
14
15
16 EXPORT_SYMBOL(open_count); //ioftracehooking.c에서 증가 시킬 예정
17 EXPORT_SYMBOL(read_count); //ioftracehooking.c에서 증가 시킬 예정
18 EXPORT_SYMBOL(close_count); //ioftracehooking.c에서 증가 시킬 예정
19 EXPORT_SYMBOL(lseek_count); //ioftracehooking.c에서 증가 시킬 예정
20 EXPORT_SYMBOL(write_count); //ioftracehooking.c에서 증가 시킬 예정
21 EXPORT_SYMBOL(read_byte); //ioftracehooking.c에서 증가 시킬 예정
22 EXPORT_SYMBOL(write_byte); //ioftracehooking.c에서 증가 시킬 예정
23 EXPORT_SYMBOL(kernel_buffer); //ioftracehooking.c에서 증가 시킬 예정
24 EXPORT_SYMBOL(given_pid); //ioftracehooking.c에서 증가 시킬 예정
25
26 typedef asmlinkage long (*sys_call_ptr_t)(const struct pt_regs *);
27 static sys_call_ptr_t *sys_call_table;
28
29 sys_call_ptr_t origin_ftrace;
30 char *system_call_table = "sys_call_table";
31
32 void make_rw(void *addr){
33     //시스템콜 테이블에 rw권한을 주는 함수
34     unsigned int level;
35     pte_t *pte = lookup_address((u64)addr, &level);
36
37     if(pte->pte &~ _PAGE_RW)
38         pte->pte |= _PAGE_RW;
39 }
40
41 void make_ro(void *addr){
42     //시스템콜 테이블에 rw권한을 뺀 함수
43     unsigned int level;
44     pte_t *pte = lookup_address((u64)addr, &level);
45
46     pte->pte = pte->pte &~ _PAGE_RW;
47 }
```

```

48
49 char * getprocessname( pid_t input )
50 {
51     //프로세스의 이름을 출력하는 함수
52     struct task_struct *task;
53     for_each_process( task )
54     {
55         if(task->pid==given_pid)
56         {
57             return(task->comm);
58         }
59     }
60     return 0;
61 }
62
63
64 static asmlinkage int ftrace(const struct pt_regs *regs){
65     //hooking 함수를 실행하는 ftrace함수
66     if(regs->di==0)
67     {
68         //ftrace함수
69         printk(KERN_INFO "[201802065] %s [%s] status [%x] read - %d / written - %d\n",getprocessname(given_pid),kernel_buffer,read_byte,write_byte);
70         printk(KERN_INFO "open[%d] close[%d] read[%d] write[%d] lseek[%d]\n", open_count,close_count,read_count,write_count,lseek_count);
71         printk(KERN_INFO "OS Assignment 2 ftrace [%d] End\n", given_pid);
72         return 0;
73     }
74     else
75     {
76         given_pid = regs->di;
77         printk(KERN_INFO "OS Assignment 2 ftrace [%d] Start\n", given_pid);
78         return 0;
79     }
80 }
81
82 static int __init hooking_init(void)
83 {
84     //hooking 함수를 적재하는 함수
85     sys_call_table = (sys_call_ptr_t *) kallsyms_lookup_name(system_call_table);
86     make_rw(sys_call_table);
87
88     origin_ftrace = sys_call_table[_NR_ftrace];
89     sys_call_table[_NR_ftrace] = (sys_call_ptr_t)ftrace; //시스템 콜 테이블을 long 타입이기에 원래에 타입 cast
90
91     //printk(KERN_INFO "Operate insmod ftracehooking.\n"); //모듈 적재를 확인하기위한 프린트문
92     return 0;
93 }
94
95 static void __exit hooking_exit(void){
96     sys_call_table[_NR_ftrace]= origin_ftrace;
97     make_ro(sys_call_table);
98     //printk(KERN_INFO "Operate rmmod ftracehooking.\n"); //모듈 해제를 확인하기위한 프린트문
99 }
100
101 module_init(hooking_init);
102 module_exit(hooking_exit);
103 MODULE_LICENSE("GPL");
104

```

위 코드에서 중점적으로 볼 사항은 결과값을 출력할 때 사용하는 변수를 선언한 부분인데 각종 카운트 변수, 바이트 카운트 변수, 파일 이름을 저장할 변수등을 EXPORT\_SYMBOL로써 사용하여 ioftacehooking.c에서 extern하여 증가 시킨다. 뿐만 아니라 ftrace함수의 역할인 pid를 인자를 받는 것 외에 그 pid를 변수에 저장하여 EXPORT\_SYMBOL로써 사용하였기 때문에 ioftacehooking.c에서 각 ftrace\_(시스템콜) 함수를 실행하였을 시 프로세스의 pid와 같을 때만 count를 가능하게 하여 테스트 코드에 사용된 시스템 콜 만 count 가능하게 하였다. 다음 사항은 ftrace함수를 정의한 부분이다. 인자값으로 const struct pt\_regs \*regs을 사용하였는데 이는 시스템 콜을 하였을 시 stack에 적재될 때 그 stack상의 reg값을 사용할 수 있는 변수이고 ftrace함수의 경우에는 di 레지스터에 pid가 저장되기 때문에 regs->di를 사용하게 되었다. 해당 값이 0일 때 ftrace를 종료하는 조건을 주어 함수를 구현하였다.

- ioftracehooking.c의 구현 코드

```
1  #include "ftracehooking.h"
2
3  extern int open_count; //EXPORT_SYMBOL에 대한 extern
4  extern int read_count; //EXPORT_SYMBOL에 대한 extern
5  extern int close_count; //EXPORT_SYMBOL에 대한 extern
6  extern int lseek_count; //EXPORT_SYMBOL에 대한 extern
7  extern int write_count; //EXPORT_SYMBOL에 대한 extern
8  extern int read_byte; //EXPORT_SYMBOL에 대한 extern
9  extern int write_byte; //EXPORT_SYMBOL에 대한 extern
10 extern char kernel_buffer[10000]; //EXPORT_SYMBOL에 대한 extern
11 extern pid_t given_pid; //EXPORT_SYMBOL에 대한 extern
12
13 pid_t getPID(void) // 유저모드에서 getpid와 같은 역할을 함
14 {
15     pid_t pid = 0;
16     struct task_struct *task=current; //이 부분으로 인해 해당 프로세스에서의 pid 가져올 수 있음
17     pid = task->pid;
18     return pid;
19 }
20
21 typedef asmlinkage long (*sys_call_ptr_t)(const struct pt_regs *);
22 static sys_call_ptr_t *sys_call_table;
23
24 sys_call_ptr_t origin_open;
25 sys_call_ptr_t origin_read;
26 sys_call_ptr_t origin_write;
27 sys_call_ptr_t origin_close;
28 sys_call_ptr_t origin_lseek;
29 char *system_call_table = "sys_call_table";
30
31 /* 이것은 ftracehooking.c에 이미 정의했기 때문에 또 한번 killed 메러님
32 void make_rw(void *addr){
33     unsigned int level;
34     pte_t *pte = lookup_address((u64)addr, &level);
35
36     if(pte->pte &~ _PAGE_RW)
37         pte->pte |= _PAGE_RW;
38 }
39 */
45 /*
46 void make_ro(void *addr){
47     unsigned int level;
48     pte_t *pte = lookup_address((u64)addr, &level);
49
50     pte->pte = pte->pte &~ _PAGE_RW;
51 }
52 */
```

```

53
54 static asmlinkage long ftrace_open(const struct pt_regs *regs)
55 {
56
57     if(given_pid==getPID())
58     {
59         //유저 공간에 위치한 문자열 메모리 공간을 커널에서 이용하기 위하여
60         //커널 공간에 버퍼를 잡고 문자열을 복사하는 과정.
61         open_count++;//프로세스의 pid와 같을 때만 카운트
62         strncpy_from_user(&kernel_buffer[0], (const char __user*)regs->di, sizeof(kernel_buffer) - 1);
63         kernel_buffer[sizeof(kernel_buffer) - 1] = '\0';
64         //printk("[+][chmod] filename :%s \n", kernel_buffer);
65     }
66     return origin_open(regs);
67 }
68 static asmlinkage long ftrace_read(const struct pt_regs *regs)
69 {
70     //printk(KERN_INFO "OS Assignment2 ftrace_read Start\n");
71     if(given_pid==getPID())
72     {
73         read_byte =read_byte+((int)regs->dx);//read 바이트를 카운트 하는 과정
74         read_count++;//프로세스의 pid와 같을 때만 카운트
75     }
76     return origin_read(regs);
77 }
78 static asmlinkage long ftrace_write(const struct pt_regs *regs)
79 {
80     //printk(KERN_INFO "OS Assignment2 ftrace_write Start\n");
81     if(given_pid==getPID())
82     {
83         write_byte =write_byte+((int)regs->dx);// write 바이트를 카운트 하는 과정
84         write_count++;//프로세스의 pid와 같을 때만 카운트
85     }
86     return origin_write(regs);
87 }
88 static asmlinkage long ftrace_lseek(const struct pt_regs *regs)
89 {
90     //printk(KERN_INFO "OS Assignment2 ftrace_lseek Start\n");
91
92     if(given_pid==getPID())
93     {
94         lseek_count++;//프로세스의 pid와 같을 때만 카운트
95     }
96     return origin_lseek(regs);
97 }

```

```

98 static asmlinkage long ftrace_close(const struct pt_regs *regs)
99 {
100     //printk(KERN_INFO "OS Assignment2 ftrace_close Start\n");
101     if(given_pid==getPID())
102     {
103         close_count++;//프로세스의 pid와 같을 때만 카운트
104     }
105     return origin_close(regs);
106 }

```



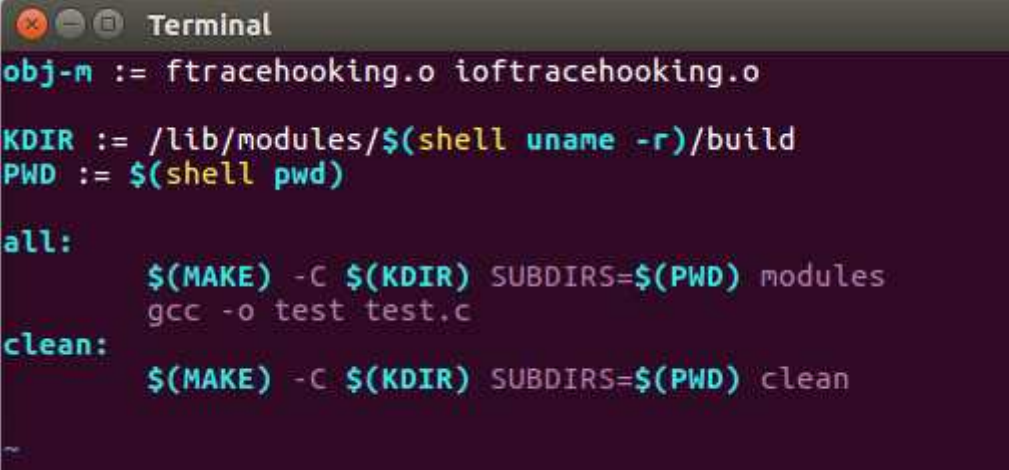
```

107
108 static int __init hooking_init(void)
109 {
110     //모듈 적재함수
111     sys_call_table = (sys_call_ptr_t *) kallsyms_lookup_name("sys_call_table");
112     //make_rw(sys_call_table); //이것은 ftracehooking.c에 이미 정의했기 때문에 꼭 하면 killed 메러닝
113     origin_open=sys_call_table[_NR_open];
114     origin_read=sys_call_table[_NR_read];
115     origin_write=sys_call_table[_NR_write];
116     origin_lseek=sys_call_table[_NR_lseek];
117     origin_close=sys_call_table[_NR_close];
118     sys_call_table[_NR_open]=ftrace_open;
119     sys_call_table[_NR_read]=ftrace_read;
120     sys_call_table[_NR_write]=ftrace_write;
121     sys_call_table[_NR_lseek]=ftrace_lseek;
122     sys_call_table[_NR_close]=ftrace_close;
123     //printk(KERN_INFO "Operate insmod ioftracehooking.\n"); //ioftrace.c 모듈 적재를 확인하기 위한 프린트문
124     return 0;
125 }
126
127 static void __exit hooking_exit(void)
128 {
129     //모듈 해제함수
130     sys_call_table[_NR_open]=origin_open;
131     sys_call_table[_NR_read]=origin_read;
132     sys_call_table[_NR_write]=origin_write;
133     sys_call_table[_NR_lseek]=origin_lseek;
134     sys_call_table[_NR_close]=origin_close;
135     //make_ro(sys_call_table); //이것은 ftracehooking.c에 이미 정의했기 때문에 꼭 하면 killed 메러닝
136     //printk(KERN_INFO "Operate rmmmod ioftracehooking.\n"); //모듈 해제를 확인하기 위한 프린트문
137 }
138
139 module_init(hooking_init);
140 module_exit(hooking_exit);
141 MODULE_LICENSE("GPL");
142

```

위의 코드에서 중점적으로 볼 사항은 ftracehooking.c에서 정의한 변수들을 이용하여 ftrace\_(시스템콜)함수들에서 사용하는 부분이다. extern을 사용하면 이를 구현할 수 있다. 또한 ftrace\_(시스템콜)함수를 기존 시스템 콜 함수를 후킹하여 대체하였을 시 시스템 전체에서 해당 시스템콜이 사용될 때 마다 count가 실행되게 되는데 이를 방지하기 위해 조건문으로 test.c를 실행하는 프로세스의 pid와 같을 때만 count하는 조건으로 해결하였다. 또한 ftrace\_open의 경우 파일이름을 추출하여 따로 선언한 변수에 저장하여야 하는데 인자값으로 const struct pt\_regs \*regs을 사용하기 때문에 위의 ftracehooking.c에서 설명한 것과 같이 구조체 변수인 레지스터들 중에 해당 값이 사용된 레지스터를 찾아야 하는데 인자를 저장하는 첫 번째 레지스터인 di레지스터를 통해 이를 가능하게 하였다. 또한 유저 공간의 파일이름을 커널 공간으로 가져와야 하는데 이는 따로 변환하는 과정이 필요하여 해당 과정을 사용한 것을 위를 보면 알 수 있다. ftrace\_wirte()함수, ftrace\_read()함수의 경우 읽거나 쓴 바이트 수를 추출하여 따로 선언한 변수에 저장해야 하고 이때 const struct pt\_regs의 3번째 인자를 저장하는 변수인 dx를 사용하여 해결하였다.

- Makefile



```
obj-m := ftracehooking.o ioftracehooking.o

KDIR := /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)

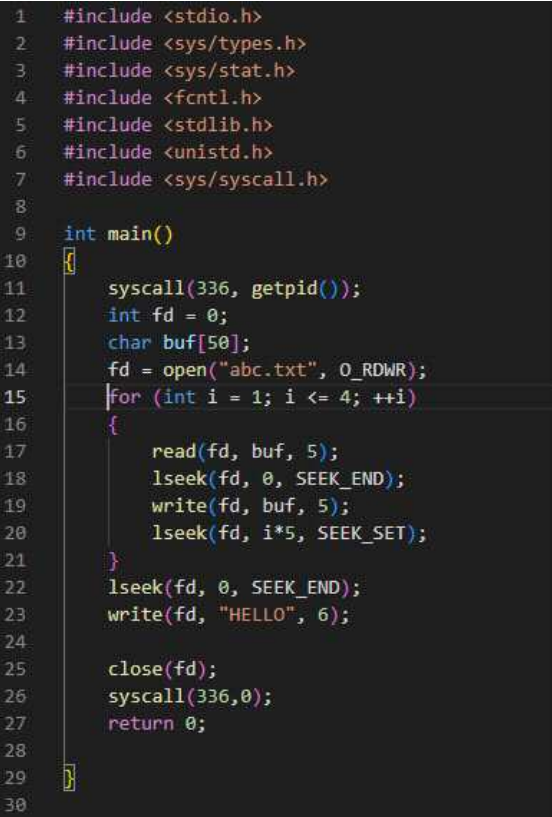
all:
    $(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules
    gcc -o test test.c

clean:
    $(MAKE) -C $(KDIR) SUBDIRS=$(PWD) clean
```

과제의 요구사항에 맞게 ftracehooking.ko파일과 ioftracehooking.ko파일을 동시에 만드는 Makefile을 구현해야하기 때문에 obj-m := ftracehooking.o ioftracehooking.o를 사용하였다.

- test파일

테스트 하기 위한 파일은 klas에서 제공받은 파일이며 코드는 아래와 같다.



```
1  #include <stdio.h>
2  #include <sys/types.h>
3  #include <sys/stat.h>
4  #include <fcntl.h>
5  #include <stdlib.h>
6  #include <unistd.h>
7  #include <sys/syscall.h>
8
9  int main()
10 {
11     syscall(336, getpid());
12     int fd = 0;
13     char buf[50];
14     fd = open("abc.txt", O_RDWR);
15     for (int i = 1; i <= 4; ++i)
16     {
17         read(fd, buf, 5);
18         lseek(fd, 0, SEEK_END);
19         write(fd, buf, 5);
20         lseek(fd, i*5, SEEK_SET);
21     }
22     lseek(fd, 0, SEEK_END);
23     write(fd, "HELLO", 6);
24
25     close(fd);
26     syscall(336,0);
27     return 0;
28 }
29
30
```

총 1번의 open, 1번의 close, 4번의 read, 5번의 write, 9번의 lseek을 사용하고 open하는 파일의 이름은 abc.txt 파일, read byte 수는 20 write byte 수는 26이다.



- 결과 도출

```
os2018202065@ubuntu:~/ftracehooking$ sudo insmod ftracehooking.ko
os2018202065@ubuntu:~/ftracehooking$ sudo insmod ioftracehooking.ko
os2018202065@ubuntu:~/ftracehooking$ ./a.out
os2018202065@ubuntu:~/ftracehooking$ sudo rmmmod ioftracehooking.ko
os2018202065@ubuntu:~/ftracehooking$ sudo rmmmod ftracehooking.ko
os2018202065@ubuntu:~/ftracehooking$ dmesg
```

명령어를 입력하고 dmesg를 확인한 결과

```
[ 491.808838] e1000: ens33 NIC Link is Down
[ 495.842752] e1000: ens33 NIC Link is Up 1000 Mbps Full Duplex, Flow Control: None
[ 497.338577] OS Assignment 2 ftrace [3595] Start
[ 497.345478] [2018202065] /a.out file[abc.txt] statas [x] read - 20 / written - 26
[ 497.345482] open[1] close[1] read[4] write[5] lseek[9]
[ 497.345484] OS Assignment 2 ftrace [3595] End
[ 497.859467] e1000: ens33 NIC Link is Down
[ 501.893024] e1000: ens33 NIC Link is Up 1000 Mbps Full Duplex, Flow Control: None
os2018202065@ubuntu:~/ftracehooking$
```

결과 값이 잘 나옴을 알 수 있다.

## ● 고찰

이번 과제를 구현하면서 발생한 이슈는 다음과 같다.

1. `__SYSCALL_DEFINE`을 사용하지 않고 `static asm` linkage를 사용하는 것
2. `ioftracehooking.c`에서 시스템콜 함수들을 후킹하여 `ftrace_(시스템콜)`함수로 대체하고 count했을 때 test파일에서 사용한 시스템콜 보다 더 많은 것들이 count되는 오류
3. `ftrace_open` 함수에서 유저 공간에서의 filename를 커널 공간으로 가지고 오는 이슈
4. 커널 모드 코드에서 pid를 가져오는 방법에 대한 이슈
5. `rmmod` 시 killed 오류가 발생하는 이슈
6. 프로세스 이름을 가져오는 방법에 관한 이슈

해결했던 방식은 다음과 같다.

1. `asm` linkage는 어셈블리 코드에서 직접 호출(링크)할 수 있다는 의미이다. 그리하여 `static asm` linkage (인자) (함수명) (`const struct pt_regs *regs`)을 사용하여야 하는데 해당 구조체는 시스템 콜을 할 시 stack에 구조체로 각 시스템 콜의 인자값이 저장되게 되는데 이를 이해하여 해결하였다.
2. 보통 이번 과제에서 count하는 시스템 콜의 경우 test파일 뿐 아니라 시스템 전체에서 필요로 할 시 실행하게 된다. 그렇기 때문에 test파일에 있어서 사용된 시스템 콜만이 count하려고 하면 test파일을 실행하는 프로세스 아이디를 비교하는 조건문을 사용하여 증가시키는 설계를 하였고 이를 통해 구현하였다.

3. `strncpy_from_user`함수를 사용하여 해결하였다.

4. `pid_t getpid(void)` // 유저모드에서 `getpid`와 같은 역할을 함

```
{
    pid_t pid = 0;
    struct task_struct *task=current;//이 부분으로 인해 해당 프로세스에서의 pid 가져올 수 있음
    pid = task->pid;
    return pid;
}
```

} 이러한 함수를 구현하여 해결하였다.

5. `ftracehooking.c`에서 시스템 콜 테이블에 `rw` 권한을 조정하는 함수인 `make_rw()`와 `make_ro`함수를 `ioftracehooking.c`에서도 사용하게 된다면 `rmmod`과정에서 변수의 종속관계 때문에 `ioftracehooking.ko`를 먼저 종료하게 될것인데 이때 시스템 콜 테이블의 `rw`권한을 뺏어 버릴 것이다. 이후 `ftracehooking.ko`를 `rmmod`하는 과정에서 뺏은 권한을 또 뺏어버리기 때문에 오류가 발생한다. 그래서 제일 마지막에 `rmmod`되는 `ftracehooking.c`에서만 사용하고 나머지는 사용하지 말아야한다.

6. 프로세스의 이름을 가져오기 위해

```
char * getprocessname( pid_t input )
```

```
{
    //프로세스의 이름을 추출하는 함수
    struct task_struct *task;
    for_each_process( task )
    {
```

```
        if(task->pid==given_pid)
        {
            return(task->comm);
        }
    }
    return 0;
} 해당 함수를 구현하여 모든 프로세스의 pid를 확인하면서 프로세스의 pid를 발견하면 해당 이름을 리턴하는 방식이다.
```