Assignment 2

수업 명: 운영체제

담당 교수님: 최상호 교수

학번: 2018202074

이름: 김상우

강의시간: 금 1,2

Introduction

해당 과제는 system call을 이용, 시스템콜에 대한 hijack을 실행하고 이를 기반으로 file tracing하는 시스템 콜을 제작하는 것이 목표입니다. 해당 과정에서 주어진 조건들 내에서 해당 파일들을 이용해 kernel을 수정하며 kernel의 systemcall level에서의 동작 및 수정이 어떤 식으로 이뤄지는지에 대한 이해를 진행하며 동시에 module을 통해 컴퓨터의 low level에서 구조적인 변경이 가능하다는 것을 배우는 것을 목적으로 하고 있다.

Conclusion & Analysis

Assignment 1. System Call table 등록(system call : ftrace 제작)

```
os2018202074@ubuntu:~/Downloads/linux-4.19.67$ vi arch/x86/entry/syscalls/syscal
l_64.tbl
```

```
333     common  io_pgetevents              __x64_sys_io_pgetevents
334     common  rseq                       __x64_sys_rseq
336     common  ftrace                     __x64_sys_ftrace

#
```

위 같은 명령어를 이용, ftrace라는 syscall을 만들고, 336이라는 시스템콜 번호를 등록한다.

```
os2018202074@ubuntu: ~/Downloads/linux-4.19.67
File  Edit  View  Search  Terminal  Help
os2018202074@ubuntu:~/Downloads/linux-4.19.67$ vi arch/x86/entry/syscalls/syscal
l_64.tbl
os2018202074@ubuntu:~/Downloads/linux-4.19.67$ vi include/linux/syscalls.h
```

```
os2018202074@ubuntu: ~/Downloads/linux-4.19.67
File  Edit  View  Search  Terminal  Help
                set_personality(personality);

        return old;
}
asmlinkage long sys_hello(void);
asmlinkage long sys_add(int, int);
asmlinkage int ftrace(pid_t pid);
#endif
```

이후 해당 명령어를 입력, 내부 파일에서 위에 언급된 sys_ftrace에 대응되는 함수에 대해 선언한다.

```
os2018202074@ubuntu: ~/Downloads/linux-4.19.67/ftrace
File  Edit  View  Search  Terminal  Help
os2018202074@ubuntu:~/Downloads/linux-4.19.67$ ls
add          crypto         include  LICENSES         net         tools
arch         Documentation  init     MAINTAINERS      README      usr
block        drivers        ipc      Makefile         samples     virt
built-in.a   firmware       Kbuild   mm               scripts     vmlinux
certs        fs             Kconfig  modules.builtin  security    vmlinux-gdb.py
COPYING      ftrace         kernel   modules.order    sound       vmlinux.o
CREDITS      hello          lib      Module.symvers   System.map
os2018202074@ubuntu:~/Downloads/linux-4.19.67$ cd ftrace
os2018202074@ubuntu:~/Downloads/linux-4.19.67/ftrace$ ls
built-in.a       ftracehooking.o  modules.builtin
ftracehooking.c  Makefile         modules.order
os2018202074@ubuntu:~/Downloads/linux-4.19.67/ftrace$ vi ftracehooking.c
```

```
#include <linux/kernel.h>
#include <linux/syscalls.h>
#include <asm/unistd.h>

SYSCALL_DEFINE1(ftrace, pid_t, pid)
{
        printk("No hijack %ld\n",pid);
        return 0;
}
~
```

이후 system call로 ftrace가 들어왔을 때를 위해 아래와 같은 위치에 SYSCALL_DEFINE1을 이용, ftrace라는 system call이 들어왔을 때, pid_t형태의 값 1개를 인자로 받고 printk로 아래와 같은 문자열을 출력하도록 한다.

```
os2018202074@ubuntu:~/Downloads/linux-4.19.67/ftrace$ vi Makefile
```

```
os2018202074@ubuntu: ~/Downloads/linux-4.19.67/ftrace
File  Edit  View  Search  Terminal  Help
obj-y := ftracehooking.o
~
~
```

```
os2018202074@ubuntu:~/Downloads/linux-4.19.67/ftrace$ cd ..
os2018202074@ubuntu:~/Downloads/linux-4.19.67$ vi Makefile
```

```
ifeq ($(KBUILD_EXTMOD),)
core-y          += kernel/ certs/ mm/ fs/ ipc/ security/ crypto/ block/ hello/ f
trace/ add/
```

이후 Makefile내를 다음과 같이 바꾸고 linux-4.19.67 파일 내 Makefile에서 core-y를 검색, 이후 뒤에 ftrace를 추가해 system call ftrace에 대해 위 파일에 대한 내용이 실행되도록 했다.

이후 make -j 8, make modules_install, make install reboot를 진행해 방금 커널 변경사항을 적용해 주었다.

Assignment 2. Code 작성

해당 과제에선 ftracehooking.h, ftracehooking.c, iotracehooking.c, Make file을 만들어야 합니다.

ftracehooking.h

*ftracehooking.h

```
#include <linux/module.h>
#include <linux/highmem.h>
#include <linux/kallsyms.h>
#include <linux/syscalls.h>
#include <asm/syscall_wrapper.h>
#include <linux/kernel.h>
#include <linux/sched.h>
#include <linux/init_task.h>

#define __NR_read 0
#define __NR_write 1
#define __NR_open 2
#define __NR_close 3
#define __NR_lseek 8
#define __NR_ftrace 336
```

```
0       common  read                    __x64_sys_read
1       common  write                   __x64_sys_write
2       common  open                    __x64_sys_open
3       common  close                   __x64_sys_close
4       common  stat                    __x64_sys_newstat
5       common  fstat                   __x64_sys_newfstat
6       common  lstat                   __x64_sys_newlstat
7       common  poll                    __x64_sys_poll
8       common  lseek                   __x64_sys_lseek
9       common  mmap                    __x64_sys_mmap
10      common  mprotect                __x64_sys_mprotect
11      common  munmap                  __x64_sys_munmap
12      common  brk                     __x64_sys_brk
"arch/x86/entry/syscalls/syscall_64.tbl" 391 lines, 15809 characters
```

위는 ftracehooking.h의 모습이다. Read, write, open, close, lseek, ftrace에 대해 System call number에 맞는 값들을 미리 #define을 통해 대입시켜 두었다.

또한 내부 코드에서의 함수 사용을 위해 필요한 헤더들을 ftracehooking.h내에 기술하였다.

Module.h : kernel modules에 대해 다루기 위함.

Kallsyms: 커널에서 추출한 심볼 정보에 접근을 위함.

Highmem.h high memory 접근을 위한 헤더 등을 사용하는 것을 볼 수 있다.

iotracehooking.c

```
#include "ftracehooking.h"

void ** syscall_table;
extern char file_name[512];
extern int cur_pid;
extern int opennum, readnum,  writenum, lseeknum, closenum;
extern size_t readlen, writelen;

static asmlinkage long (*real_open)(const struct pt_regs* regs);
static asmlinkage long ftrace_open(const struct pt_regs* regs){
        if(get_current()->pid == cur_pid){
                char __user * fileread = (char*)regs->di;
                opennum +=1;
                strncpy_from_user(file_name, fileread, sizeof(file_name));
        }
        return real_open(regs);
}
```

해당 코드는 ftracehooking.h를 기반으로 헤더를 받아오고 있다. 이후 system call table을 받아올 syscall_table과 파일이름을 저장할 file_name 변수를 선언하고, 현재 pid를 담는 변수 cur_pid를 선언한다. 추가적으로 각 함수의 횟수와 read, write에 대한 byte값을 받아올 변수도 선언한다.

위 변수들은 extern을 통해 외부(ftracehooking.c)와 연동되어 처리되게 될 것이다.

이후 5가지 system call(open, read, write, lseek, close)에 대해 기존의 systemcall에 대한 함수를 저장할 static asmlinkage long real_(system call명)을 선언를 선언하고, 이들을 대체할 함수를 static asmlikage long ftrace_(system call명) 로서 선언했다.

위의 경우 원래 systemcall open에 대해 반응하는 함수를 저장할 real_open을 선언하고 이를 대체할 ftrace_open를 선언, 해당 함수가 regs를 통해 들어온 process와 현재의 process가 같은지 확인하고 맞을 경우, 현재 file명을 받고, open 실행 횟수를 1 높인다. 이후 real_open을 반환한다.

```
static asmlinkage long (*real_read)(const struct pt_regs* regs);
static asmlinkage long ftrace_read(const struct pt_regs* regs){
        if(get_current()->pid == cur_pid){
                readnum +=1;
                readlen += (size_t)regs->dx;
        }
        return real_read(regs);
}

static asmlinkage long (*real_write)(const struct pt_regs* regs);
static asmlinkage long ftrace_write(const struct pt_regs* regs){
        if(get_current()->pid == cur_pid){
                writenum +=1;
                writelen += (size_t)regs->dx;
        }
        return real_write(regs);
}

static asmlinkage long (*real_lseek)(const struct pt_regs* regs);
static asmlinkage long ftrace_lseek(const struct pt_regs* regs){
        if(get_current()->pid == cur_pid){
                lseeknum +=1;
        }
        return real_lseek(regs);
}
```

```
static asmlinkage long ftrace_close(const struct pt_regs* regs){
        if(get_current()->pid == cur_pid){
                closenum +=1;
        }
        return real_close(regs);
}
```

위들도 마찬가지로, system call read, write, lseek, close에 대해 원래 사용되던 함수를 저장하고 대체할 함수들을 선언하고 있다. 모든 대체 함수들이 regs를 통해 들어온 pid와 현재 실행중인 process의 pid가 같은지 확인하고 맞다면 해당 함수를 실행한 횟수에 대한 변수를 1 상승시키고 최종적으로 원래 system call에 사용된 함수에 regs를 그대로 넘긴 상태로 반환한다. 이때, read와 write의 경우 text의 길이를 알기 위해 pt_regs regs의 dx값을 더하게 된다.

```
static int __init io_hooking_init(void){
        syscall_table = (void **)kallsyms_lookup_name("sys_call_table");

        real_open = syscall_table[__NR_open];
        syscall_table[__NR_open] = ftrace_open;
        real_read = syscall_table[__NR_read];
        syscall_table[__NR_read] = ftrace_read;
        real_write = syscall_table[__NR_write];
        syscall_table[__NR_write] = ftrace_write;
        real_lseek = syscall_table[__NR_lseek];
        syscall_table[__NR_lseek] = ftrace_lseek;
        real_close = syscall_table[__NR_close];
        syscall_table[__NR_close] = ftrace_close;
        return 0;
}
static void __exit io_hooking_exit(void){
        syscall_table[__NR_open] = real_open;
        syscall_table[__NR_read] = real_read;
        syscall_table[__NR_write] = real_write;
        syscall_table[__NR_lseek] = real_lseek;
        syscall_table[__NR_close] = real_close;
}
module_init(io_hooking_init);
module_exit(io_hooking_exit);
MODULE_LICENSE("GPL");
```

이후 해당 함수들에 대한 대체가 이뤄지기 전 처리를 위해 __init io_hooking_init를 사용, syscall table내에 kallsyms_lookup_name을 통해 기존에 사용하던 sys_call_table, 즉 system call에 대한 표를 받아온다. 이후 real_(system call명) 내에 원래 위치에 있던 함수를 syscall_table[system call number]를 통해 저장하고, syscall_table[system call_number]의 위치에 대체할 함수를 넣는 것을 볼 수 있다.

이는 __exit io_hooking_exit를 통해 원래대로 돌릴 필요가 있다. 이를 위해 syscall_table[system call_number] 위치에 이전에 real_(system call명)에 저장했던 원래 함수를 다시 넣어준다.

이후 module_init, module_exit를 통해 위 코드 기반의 모듈의 각 생성/종료때 각 함수를 실행한다.

Module_license("GPL")을 통해 모듈의 라이선스는 GPL이 된다.

ftracehooking.c

```
#include "ftracehooking.h"

void ** syscall_table;  //save system call
void *real_ftrace;
char file_name[512];
int pid;
int cur_pid = 0;
int opennum, readnum, writenum, lseeknum, closenum = 0;
size_t readlen, writelen = 0;
```

해당 코드는 ftracehooking.h를 기반으로 헤더를 받아오고 있다. 이후 system call table을 받아올 syscall_table과 원래 ftrace함수를 저장할 변수, 파일이름을 저장할 file_name 변수를 선언하고, 들어온 pid/현재 pid를 담는 변수 pid/cur_pid를 선언한다. 추가적으로 각 함수의 횟수와 read, write에 대한 byte값을 받아올 변수도 선언하고 0으로 초기화한다.

```
EXPORT_SYMBOL(cur_pid);

EXPORT_SYMBOL(file_name);

EXPORT_SYMBOL(opennum);
EXPORT_SYMBOL(readnum);
EXPORT_SYMBOL(writenum);
EXPORT_SYMBOL(lseeknum);
EXPORT_SYMBOL(closenum);

EXPORT_SYMBOL(readlen);
EXPORT_SYMBOL(writelen);

#define __NR_ftrace 336
```

이후 ioftracehooking.c에서도 사용될 위 변수들인, cur_pid(현재 프로세스 pid확인), file_name(실행중인 file명 확인), systemcall명+num(system call 호출 수), readlen, writelen(읽고 쓰는 byte 수)에 대하여 EXPORT_SYMBOL을 통해 처리한다. 이는 위에서 언급된 Extern을 통해 ioftracehooking.c에서의 count등에서 처리되게 될 것이다.

```
void make_rw(void *addr)
{
        unsigned int level;
        pte_t *pte = lookup_address((u64)addr, &level);
        if(pte->pte &~ _PAGE_RW)
                pte->pte |= _PAGE_RW;
}

void make_ro(void *addr)
{
        unsigned int level;
        pte_t * pte = lookup_address((u64)addr, &level);
        pte->pte = pte->pte &~ _PAGE_RW;
}
```

이후 addr이 속해 있는 페이지에 읽기 및 쓰기 권한을 부여하는 햠수 make_rw를 위와 같이 선언하고 이를 통해 system call table에 대해 쓰기 권한을 취득하여 해당 model이 hijack할 수 있게 한다. Make_ro를 통해서는 이를 원래대로 돌린다.

```
static asmlinkage int hook_ftrace(const struct pt_regs *regs){
        pid = regs->di;
        if(pid !=0){
                cur_pid = (int)pid;
                printk("OS Assignment 2 ftrace [%d] Start \n", pid);
                opennum = 0;
                readnum = 0;
                writenum = 0;
                lseeknum = 0;
                closenum = 0;
                readlen = 0;
                writelen = 0;
        }
        else{
                printk("[2018202074] /%s file[%s] stats [x] read - %lu / written - %lu ", get_current()->comm, file_name,
readlen, writelen);
                printk("open[%d] close[%d] read[%d] write[%d] lseek[%d]\n", opennum, closenum, readnum, writenum, lseeknum);
                printk("OS Assignment 2 ftrace [%d] End\n", cur_pid);
                cur_pid = 0;
        }
        return 0;
}
```

이후 ftrace를 대체할 함수를 선언하였다. Regs를 통해 pid를 추출해내고 기존에 제시된 종료시 system call에 0 값을 넣고 호출한다는 기준에 맞춰 해당 값이 0이 아닐 시 각 함수들을 0으로 초기화함과 동시에 printk로 현재의 pid를 알맞은 양식으로 출력하게 하며, ioftracehooking.c에서 사용하기 위해 현재의 process의 pid를 별개로 저장한다.

만약 pid가 0, 즉 종료를 위해 system call에 0값이 들어왔다면 ioftracehooking.c를 바탕으로 처리되어 값을 받아온 변수들에 대해 양식에 맞춰 출력할 수 있게 해주었다. 동시에 pid의 변경을 위해 cur_pid는 0이 된다.

```
static int __init hooking_init(void)
{
        syscall_table = (void **) kallsyms_lookup_name("sys_call_table");
        make_rw(syscall_table);
        real_ftrace = syscall_table[__NR_ftrace];
        syscall_table[__NR_ftrace] = hook_ftrace;
        return 0;
}
static void __exit hooking_exit(void)
{
        syscall_table[__NR_ftrace] = real_ftrace;
        make_ro(syscall_table);
}
module_init(hooking_init);
module_exit(hooking_exit);
MODULE_LICENSE("GPL");
```

이후 해당 함수들에 대한 대체가 이뤄지기 전 처리를 위해 __init hooking_init를 사용, syscall table 내에 kallsyms_lookup_name을 통해 기존에 사용하던 sys_call_table, 즉 system call에 대한 표를 받아온다. 이후 real_ftrace 내에 원래 위치에 있던 함수를 syscall_table[system call number]를 통해 저장하고, syscall_table[__NR_ftrace]의 위치에 대체할 함수를 넣는 것을 볼 수 있다.

이는 __exit hooking_exit를 통해 원래대로 돌릴 필요가 있다. 이를 위해 syscall_table[__NR_ftrace] 위치에 이전에 real_ftrace에 저장했던 원래 함수를 다시 넣어준다.

이후 module_init, module_exit를 통해 위 코드 기반의 모듈의 각 생성/종료때 각 함수를 실행한다.

Module_license("GPL")을 통해 모듈의 라이선스는 GPL이 된다.

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

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

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

Make file의 경우 ftracehooking.ko 파일과 iotracehooking.ko 파일이 동시에 생성될 필요가 있다. 이를 위해 위와 같은 코드를 작성, ftracehooking.o와 iotracehooking.o에 대해 obj-m, 즉 모듈로서 선언을 하고 KDIR을 이용, module로서의 동작이 가능하도록 kernel을 uname -r로서 확인하고 kernel내 build 위치에 Make 명령어를 쓸수 있도록 해두었으며 내부에 접근하는 데 있어 pwd를 이용해 가능하도록 하였다.

Assignment 3. Test & Result

```
  LD [M]  sound/usb/usx2y/snd-usb-us122l.ko
  LD [M]  sound/usb/usx2y/snd-usb-usx2y.ko
  LD [M]  sound/x86/snd-hdmi-lpe-audio.ko
os2018202074@ubuntu:~/Downloads/linux-4.19.67$ sudo make modules_install
```

```
Found linux image: /boot/vmlinuz-4.15.0-29-generic
Found initrd image: /boot/initrd.img-4.15.0-29-generic
Found memtest86+ image: /boot/memtest86+.elf
Found memtest86+ image: /boot/memtest86+.bin
done
os2018202074@ubuntu:~/Downloads/linux-4.19.67$ reboot
```

컴파일을 통해 다시금 위에서 진행된 처리들에 대해 업데이트 한다.

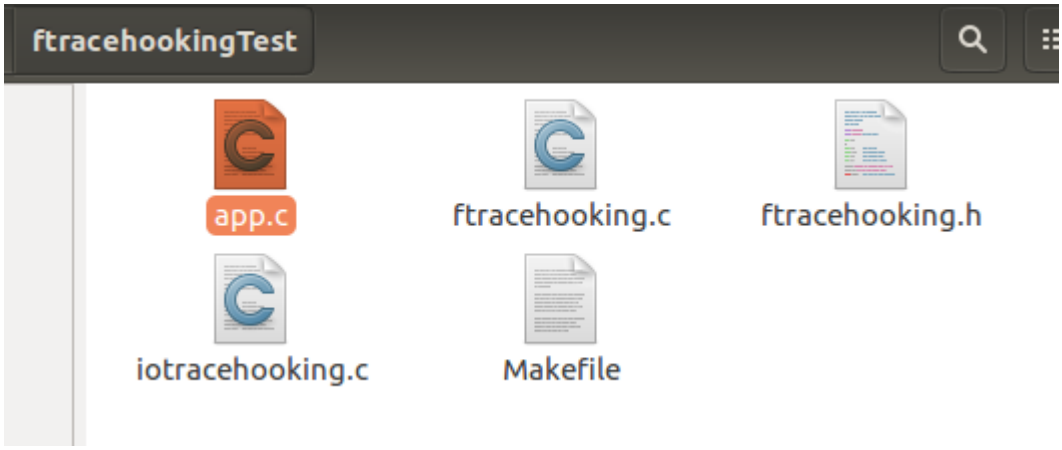
(make -j 8, sudo make modules_install, sudo make install, reboot)

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/syscall.h>

int main()
{
        syscall(336, getpid());
        int fd = 0;
        char buf[50];
        fd = open("abc.txt", O_RDWR);
        for (int i = 1; i <= 4; ++i)
        {
                read(fd, buf, 5);
                lseek(fd, 0, SEEK_END);
                write(fd, buf, 5);
                lseek(fd, i*5, SEEK_SET);
        }
        lseek(fd, 0, SEEK_END);
        write(fd, "HELLO", 6);
        close(fd);
        syscall(336,0);
        return 0;

}
```

이후 강의자료로 제시된 c파일을 가져왔다. 이름은 app.c로 지정하였다.




```
os2018202074@ubuntu:~/ftracehookingTest$ ls
app.c  ftracehooking.c  ftracehooking.h  iotracehooking.c  Makefile
os2018202074@ubuntu:~/ftracehookingTest$ sudo make
make -C /lib/modules/4.19.67-os2018202074/build SUBDIRS=/home/os2018202074/ftrac
ehookingTest modules
make[1]: Entering directory '/home/os2018202074/Downloads/linux-4.19.67'
  CC [M]  /home/os2018202074/ftracehookingTest/ftracehooking.o
  CC [M]  /home/os2018202074/ftracehookingTest/iotracehooking.o
  Building modules, stage 2.
  MODPOST 2 modules
  CC      /home/os2018202074/ftracehookingTest/ftracehooking.mod.o
  LD [M]  /home/os2018202074/ftracehookingTest/ftracehooking.ko
  CC      /home/os2018202074/ftracehookingTest/iotracehooking.mod.o
  LD [M]  /home/os2018202074/ftracehookingTest/iotracehooking.ko
make[1]: Leaving directory '/home/os2018202074/Downloads/linux-4.19.67'
gcc -o app app.c
os2018202074@ubuntu:~/ftracehookingTest$ ls
app              ftracehooking.ko     iotracehooking.c      iotracehooking.o
app.c            ftracehooking.mod.c  iotracehooking.ko     Makefile
ftracehooking.c  ftracehooking.mod.o  iotracehooking.mod.c  modules.order
ftracehooking.h  ftracehooking.o      iotracehooking.mod.o  Module.symvers
os2018202074@ubuntu:~/ftracehookingTest$
```

내부에 해당 파일들을 넣고 sudo make를 진행, 위와 같이 실행된 후 파일들이 생겼으며 iotracehooking.ko와 ftracehooking.ko가 동시에 생김을 확인했다.

아래는 ./app을 실행한 결과이다.

```
os2018202074@ubuntu:~/ftracehookingTest$ ./app
os2018202074@ubuntu:~/ftracehookingTest$ dmesg
```

```
[11874.316086] No hijack 11522
[11874.316096] No hijack 0
os2018202074@ubuntu:~/ftracehookingTest$
```

기존에 지정해둔 ftrace함수대로 생성되는 것을 확인했다.

```
os2018202074@ubuntu:~/ftracehookingTest$ sudo insmod ftracehooking.ko
os2018202074@ubuntu:~/ftracehookingTest$ sudo insmod iotracehooking.ko
os2018202074@ubuntu:~/ftracehookingTest$ ./app
os2018202074@ubuntu:~/ftracehookingTest$ dmesg
```

Sudo insmod ftracehooking.ko sudo insmod iotracehooking.ko를 실행해 hijacking이 가능하게 만든 후 다시 똑같은 실행을 반복해보았다.

```
[11874.316086] No hijack 11522
[11874.316096] No hijack 0
[12047.796144] OS Assignment 2 ftrace [11537] Start
[12047.796154] [2018202074] /app file[abc.txt] stats [x] read - 20 / written - 26
[12047.796155] open[1] close[1] read[4] write[5] lseek[9]
[12047.796156] OS Assignment 2 ftrace [11537] End
os2018202074@ubuntu:~/ftracehookingTest$
```

기대한 결과값이 나오는 것을 확인할 수 있었다.

```
os2018202074@ubuntu:~/ftracehookingTest$ sudo rmmod iotracehooking.ko
os2018202074@ubuntu:~/ftracehookingTest$ sudo rmmod ftracehooking.ko
os2018202074@ubuntu:~/ftracehookingTest$ ./app
os2018202074@ubuntu:~/ftracehookingTest$ dmesg
```

이후 sudo rmmod iotracehooking.ko, sudo rmmod ftracehooking.ko를 통해 해당 설정을 없애고 다시 실행해보았다.

```
[11874.316086] No hijack 11522
[11874.316096] No hijack 0
[12047.796144] OS Assignment 2 ftrace [11537] Start
[12047.796154] [2018202074] /app file[abc.txt] stats [x] read - 20 / written - 26
[12047.796155] open[1] close[1] read[4] write[5] lseek[9]
[12047.796156] OS Assignment 2 ftrace [11537] End
[12209.580375] No hijack 11561
[12209.580385] No hijack 0
os2018202074@ubuntu:~/ftracehookingTest$
```

다시 원해 ftrace함수로 돌아온 것을 확인할 수 있었다.

app.c (~/ftracehookingTest) - gedit

Open Save

```c
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/syscall.h>

int main()
{
        syscall(336, getpid());
        int fd = 0;
        char buf[50];
        fd = open("abc.txt", O_RDWR);
        for (int i = 1; i <= 5; ++i)
        {
                read(fd, buf, 5);
                lseek(fd, 0, SEEK_END);
                write(fd, buf, 5);
                lseek(fd, i*6, SEEK_SET);
        }
        lseek(fd, 0, SEEK_END);
        write(fd, "HELLO", 3);
        close(fd);
        syscall(336,0);
        return 0;
```

```
[ 1482.873873] OS Assignment 2 ftrace [5092] Start
[ 1482.873883] [2018202074] /app file[abc.txt] stats [x] read - 25 / written - 28
[ 1482.873884] open[1] close[1] read[5] write[6] lseek[11]
[ 1482.873884] OS Assignment 2 ftrace [5092] End
os2018202074@ubuntu:~/ftracehookingTest$
```

코드가 달라졌을 때도 예상한 값이 나오는 것을 확인할 수 있었다.

고찰

이번 과제를 통해 실제로 system call수준에서 작업해보면서 가상환경이 돌아갈 수 없게 되기도 하고, 그 과정에서 VMware와 Ubuntu의 재설치를 진행하며 온라인 환경에서 가상환경을 구축하는 법을 다시금 복습해 볼 수 있었습니다. 또한 module에 대해 직접 건들이거나 system call을 직접 만들어보면서 kernel이 system call을 처리하는 데 있어서 어떠한 방식을 사용하는 지 어느 정도 알 수 있었습니다. 또한, 만든 함수를 스스로 하이재킹 함으로서 해당 구조에 대해 좀 더 확실하게 알 수 있었으며 동시에 수정이 어떻게 이뤄지는지도 알 수 있는 좋은 실습이었습니다.

Reference

강의자료 참고

22-2_OSLab_04_Systemcall.pdf

22-2_OSLab_05_Module_Programming__Wrapping.pdf

22-2_OSLab_06_Task_Management.pdf

외부자료 참고

https://stackoverflow.com/questions/2103315/linux-kernel-system-call-hooking-example

https://junshim.github.io/linux%20kernel%20study/Add_a_New_System_Call/

https://www.bhral.com/post/waios-dump_stack-%ED%95%A8%EC%88%98-%EA%B5%AC%ED%98%84