2022학년도 2학기

운영체제

-REPORT-

수업명	운영체제
과제 이름	assignment2
담당 교수님	최상호 교수님
학번	2018204058
이름	김민교

1. Introduction :

시스템 콜(system call)은 응용 프로그램의 요청에 따라 커널에 접근하기 위한 인터페이스이다. 본 과제2에서는 시스템콜을 정의하고, 정의한 시스템콜을 후킹할 것이다. 또한 기존의 있던 파일 입출력과 관련된 open, read, write, lseek, close 시스템콜을 후킹한다. 이를 통해서 파일 입출력이 어떻게 커널에서 수행되는지 알고, pt_reg 인자를 통해 각 시스템콜의 매개변수에 접근하는 것을 배운다.

2. Conclusion & Analysis :

[과제 요구사항]

```
[ 4445.322428] OS Assignment 2 ftrace [4816] Start
[ 4445.322491] [2018204058] a.out file[abc.txt] start [x] read - 20 / written - 26
[ 4445.322492] open[1] close[1] read[4] write[5] lseek[9]
[ 4445.322492] OS Assignment 2 ftrace [4816] End
```

Ftrace 시스템콜

```
os2018204058@ubuntu: ~/Downloads/linux-4.19.67
os2018204058@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
#
# x32-specific system call numbers start at 512 to avoid cache impact
```

시스템 콜 테이블에 이름 ftrace, 시스템콜 번호 336을 등록한다.

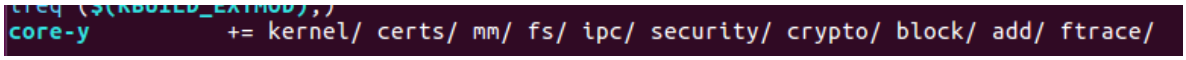
```
os2018204058@ubuntu:~/Downloads/linux-4.19.67$ vi include/linux/syscalls.h
```

```
        if (personality != 0xffffffff)
                set_personality(personality);

        return old;
}

asmlinkage long sys_add(int,int);
asmlinkage int ftrace(pid_t pid);
-- INSERT --                                              1297,34       99%
```

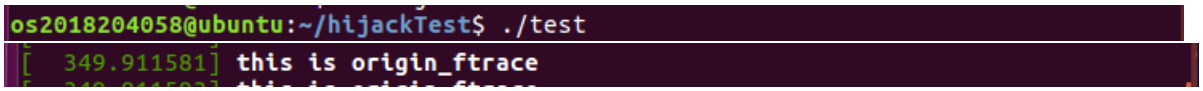
```
os2018204058@ubuntu:~/Downloads/linux-4.19.67$ mkdir ftrace
```

```
os2018204058@ubuntu:~/Downloads/linux-4.19.67$ ls
add             Documentation  Kbuild           modules.order   tags
arch            drivers        Kconfig          Module.symvers  tools
block           firmware       kernel           net             usr
built-in.a      freace         lib              README          virt
certs           fs             LICENSES         samples         vmlinux
COPYING         ftrace         MAINTAINERS      scripts         vmlinux-gdb.py
CREDITS         include        Makefile         security        vmlinux.o
crypto          init           mm               sound
cscope.out      ipc            modules.builtin  System.map
```

ftrace 폴더를 생성한다.



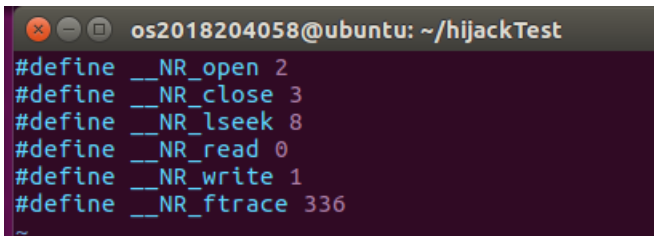
시스템 콜 처리 함수를 간단히 구현했다. 그리고나서 커널을 재 컴파일 하고 reboot하였다.



test 파일을 실행했다. ftrace 시스템콜이 잘 작동했다.

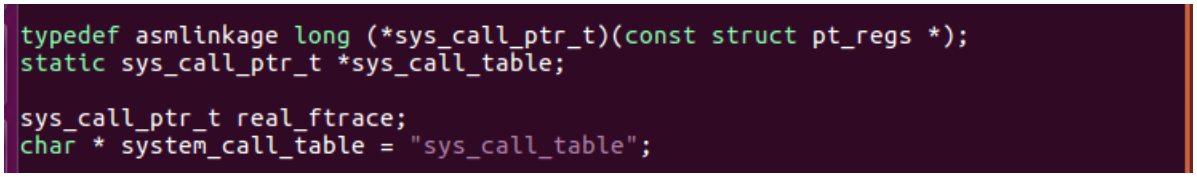
[ftrace 시스템콜 hijack]

<ftracehooking.h>



원본 시스템 콜의 번호를 선언하였다.

<ftracehooking.c>



asmlinkage는 어셈블리 코드에서 직접 호출할 수 있다는 의미다. 원본 ftrace 함수의 주소를 저장할 변수를 선언하고 system_calll_table에 접근할 변수를 선언했다.

```
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;
}
```

make_rw, make_ro는 addr이 속해 있는 페이지의 읽기 및 쓰기 권한을 부여하는 함수이다. 기본적으로, system call table은 쓰기 권한이 존재하지 않는다. make_rw를 호출하여 쓰기 권한이 없는 system call table에 쓰기 권한을 부여하고 make_ro를 호출하여 쓰기 권한을 회수한다.

```
static asmlinkage int hooking_ftrace(const struct pt_regs *regs){

        pid_t given_pid = regs->di; //get pid

        if(given_pid == 0)
        {

                //if syscall_num == 0, trace End, and print trace result
                printk(KERN_INFO "[2018204058] %s file[%s] start [x] read - %ld / written - %ld\n",__FILE__,_file_name,_read_bytes,_written_bytes);
                printk(KERN_INFO "open[%d] close[%d] read[%d] write[%d] lseek[%d]\n",_open_count,_close_count,_read_count,_write_count,_lseek_count);
                printk(KERN_INFO "OS Assignment 2 ftrace [%ld] End\n",save_pid);
        }
        else{
                printk(KERN_INFO "OS Assignment 2 ftrace [%ld] Start\n",given_pid );
                _read_bytes=0;
                _written_bytes=0;
                _open_count=0;
                _close_count=0;
                _read_count=0;
                _write_count=0;
                _lseek_count=0;
                save_pid = given_pid;
        }
        return 0;
}
```

과제를 수행하기 위해 짠 코드이다. 시스템콜을 호출하면 인자값으로 pid값이 넘어온다. 이 pid가 0이면 trace를 종료하고 입출력 함수에 대해 커널프린트를 한다. Pid의 값은 보존되어야 하기 때문에 또 다른 char형 변수에다가 저장해놓았다.

```c
static int __init hooking_init(void)
{
        /*Find system call table */
        sys_call_table = (sys_call_ptr_t *) kallsyms_lookup_name(system_call_table);

        /*
         * Change permission of the page of system call table
         * to both readable and writeable
         */

        make_rw(sys_call_table);
        real_ftrace = sys_call_table[__NR_ftrace];
        sys_call_table[__NR_ftrace] =(sys_call_ptr_t)hooking_ftrace;

        printk(KERN_INFO "Operate insmod ftracehooking. \n");
        return 0;
}


static void __exit hooking_exit(void)
{
        sys_call_table[__NR_ftrace] = real_ftrace;
        /* Recover the page's permission (i.e read-only)*/
        make_ro(sys_call_table);
        printk(KERN_INFO "Operate rmmod ftracehooking.\n");
}

module_init(hooking_init);
module_exit(hooking_exit);
MODULE_LICENSE("GPL");
```

모듈 적재시 호출되는 함수와 모듈 해제 시 호출되는 함수들이다.

close 함수의 원형

```c
asmlinkage long sys_close(unsigned int fd);
```

open 함수의 원형

```c
asmlinkage long sys_open(const char __user *filename,
                         int flags, umode_t mode);
```

write 함수의 원형

```c
asmlinkage long sys_write(unsigned int fd, const char __user *buf,
                          size_t count);
```

read 함수의 원형

```c
asmlinkage long sys_read(unsigned int fd, char __user *buf, size_t count);
```

lseek 함수의 원형

```c
asmlinkage long sys_lseek(unsigned int fd, off_t offset,
                          unsigned int whence);
```

원형 함수를 보고 pt_regs에서 어떤 변수를 가져와야할 지 생각한다.

```
0       common  read                    __x64_sys_read
1       common  write                   __x64_sys_write
2       common  open                    __x64_sys_open
3       common  close                   __x64_sys_close
8       common  lseek                   __x64_sys_lseek
```

파일 입출력 함수들의 시스템콜 번호를 찾는다.

<iotracehooking.c>

```
os2018204058@ubuntu:~/hijackTest$ vi iotracehooking.c
```

os2018204058@ubuntu: ~/hijackTest

```c
#include <linux/module.h>
#include <linux/highmem.h>
#include <linux/kallsyms.h>
#include <linux/syscalls.h>
#include <asm/syscall_wrapper.h>
#include <linux/init.h>
#include "ftracehooking.h"


typedef asmlinkage long (*sys_call_ptr_t)(const struct pt_regs *);
static sys_call_ptr_t *sys_call_table;

//original file read-write functions
sys_call_ptr_t origin_open;
sys_call_ptr_t origin_read;
sys_call_ptr_t origin_write;
sys_call_ptr_t origin_lseek;
sys_call_ptr_t origin_close;

char *system_call_table = "sys_call_table";

char _file_name[50]={0,}; //filename limit 50 characters.

long  _read_bytes;
long _written_bytes;

int _read_count;
int _open_count;
int _close_count;
int _write_count;
int _lseek_count;

//EXPORT_SYMBOL for ftracehooking function print values
EXPORT_SYMBOL(_file_name);
EXPORT_SYMBOL(_read_bytes);
EXPORT_SYMBOL(_written_bytes);
EXPORT_SYMBOL(_read_count);
EXPORT_SYMBOL(_open_count);
EXPORT_SYMBOL(_write_count);
EXPORT_SYMBOL(_lseek_count);
EXPORT_SYMBOL(_close_count);
```

EXPORT_SYMBOL 매크로를 활용하여, Ftracehooking함수가 참조할 수 있게 하였다.

```
// read write permission to PAGE (addr belongs)
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;
}

// get back read write permission to PAGE (addr belongs)
void make_ro(void *addr){
        unsigned int level;
        pte_t *pte = lookup_address((u64)addr, &level);

        pte-> pte = pte->pte &~ _PAGE_RW;
}
```

make_rw, make_ro는 addr이 속해 있는 페이지의 읽기 및 쓰기 권한을 부여하는 함수이다. 기본적으로, system call table은 쓰기 권한이 존재하지 않는다. make_rw를 호출하여 쓰기 권한이 없는 system call table에 쓰기 권한을 부여하고 make_ro를 호출하여 쓰기 권한을 회수한다.

```
static asmlinkage long ftrace_open(const struct pt_regs *regs)
{
        char __user *filename = (char*) regs->di; // get filename
        // copy using strncpy_from_user
        long copied = strncpy_from_user(_file_name,filename,sizeof(_file_name));
        _open_count+=1;
        return (origin_open(regs)); //origin open  system call

}

static asmlinkage long ftrace_read(const struct pt_regs *regs)
{
        _read_count+=1;
        _read_bytes += regs->dx;
        return (origin_read(regs)); //origin read system call
}

static asmlinkage long ftrace_write(const struct pt_regs *regs)
{
        _write_count+=1;
        _written_bytes+=regs->dx;
        return (origin_write(regs)); //origin write system call
}

static asmlinkage long ftrace_lseek(const struct pt_regs *regs)
{
        _lseek_count+=1;
        return (origin_lseek(regs)); //origin lseek system call
}

static asmlinkage long ftrace_close(const struct pt_regs *regs)
{
        _close_count+=1;
        return (origin_close(regs)); //origin close system call
}
```

open, read, write, lseek, close 시스템 콜을 후킹하는 함수이다. 함수 안에서 각각의 count를 증가시키고, read와 write bytes를 기록한다. 이는 나중에 ftracehooking.c에서 사용하게 될 것이다.

```c
static int __init init(void){
        sys_call_table = (sys_call_ptr_t*) kallsyms_lookup_name(system_call_table);
        make_rw(sys_call_table);

        origin_open = sys_call_table[__NR_open];
        origin_read = sys_call_table[__NR_read];
        origin_write = sys_call_table[__NR_write];
        origin_close = sys_call_table[__NR_close];
        origin_lseek = sys_call_table[__NR_lseek];

        sys_call_table[__NR_open] = ftrace_open;
        sys_call_table[__NR_read] = ftrace_read;
        sys_call_table[__NR_write] = ftrace_write;
        sys_call_table[__NR_close] = ftrace_close;
        sys_call_table[__NR_lseek] = ftrace_lseek;

        printk(KERN_INFO "Operate insmod iotracehooking.\n");

        return 0;
}

static void __exit my_exit(void){

        // sys_call_table have to point original system call.
        sys_call_table[__NR_open] = origin_open;
        sys_call_table[__NR_read] = origin_read;
        sys_call_table[__NR_write] = origin_write;
        sys_call_table[__NR_close] = origin_close;
        sys_call_table[__NR_lseek] = origin_lseek;
        make_ro(sys_call_table);
        printk(KERN_INFO "Operate rmmod iotracehooking.\n");

}

module_init(init);
module_exit(my_exit);
MODULE_LICENSE("GPL");
```

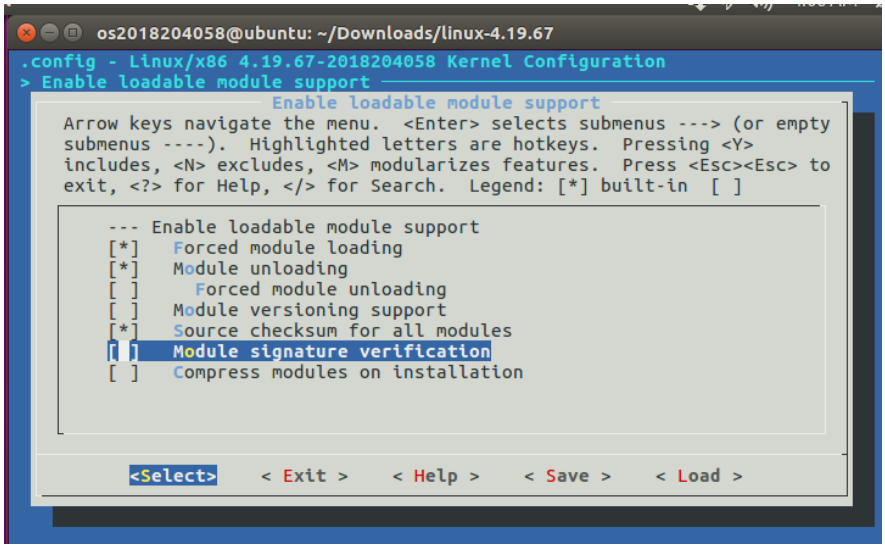
모듈 적재시 호출되는 함수와 모듈 해제 시 호출되는 함수들이다. 후킹할 시스템 콜로 바꾸기도 하고 후킹했던 시스템을 원래대로 복원한다

<Makefile>

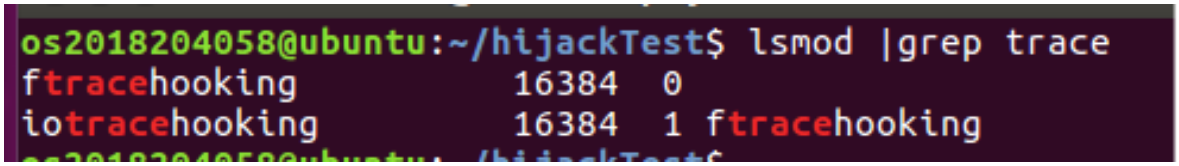
os2018204058@ubuntu: ~/hijackTest

```
CONFIG_MODULE_SIG=n
obj-m += ftracehooking.o
obj-m += iotracehooking.o

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

all:
        $(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules

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

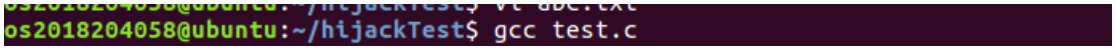
CONFIG_MODULE_SIG=n을 넣은 이유는 모듈을 삽입할 때 verification문제가 발생했었기 때문이다.



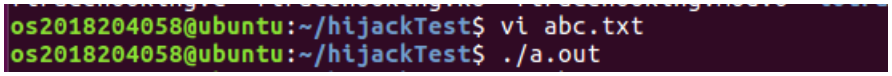
Module signature verification까지 해제 해주고 커널 컴파일 리부팅 해서야 모듈 적재할 때 오류가 나지 않았다.



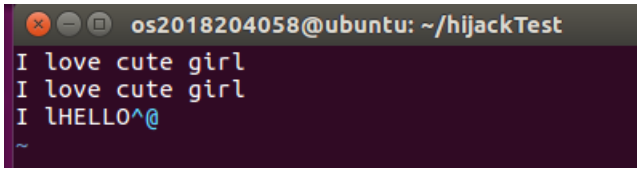
sudo insmod 명령어를 이용해 모듈을 설치하였다.



test.c 파일을 이용했다.



abc.txt를 생성했다



고찰

과제를 수행하면서 커널 모듈식 프로그래밍이 정말 편하다는 것을 느꼈다. 시스템콜 함수를 바꾸거나 추가하려면 재컴파일 해야하는데, 컴파일 시간이 굉장히 오래걸린다. 하지만 모듈 프로그래밍을 통해 컴파일 없이 자신이 원하는 시스템을 적재할 수 있었다. 이 원리는 알고 보니 쉬웠다. 함수의 주소를 system call table에 넣기만 하면 되었다. 컴퓨터 세계에서 주소가 정말 중요하다는 걸 깨달았다. 또한

데이터 교환함수를 써야 사용자 영역의 데이터를 커널 영역으로 복사 할 수 있다. 운영체제 이론수업에서 배웠듯이 정말로 사용자 영역과 커널 영역이 완전히 분리되어 있다는 것을 느꼈다.

pt_regs에 대하여 아예 몰랐는데, 과제를 하면서 조금 알게되었다. 매개변수들이 처음 세 레지스터에 저장되는 것을 알았다. 구글링을 많이 해봤는데 아직 감이 잘 안온다. pt_regs에 대해 더 공부를 하여 앞으로 있을 과제에 대비할 것이다.