KWANGWOON UNIVERSITY

운영체제
Assignment 2 과제

수업 명 : 운영체제
과제 이름 : assignment2
담당 교수님 : 최상호 교수님
학 번 : 2019202005
이 름 : 남종식

Introduction

이번 과제는 특정 PID에 대하여 커널 코드 내의 파일에 관한 시스템콜을 추적하는 툴을 작성하는 과제입니다. 즉 file tracing하는 툴인 ftrace를 작성해야 합니다. 추적할 시스템 콜은 총 5가지로 open, read, write, lseek, close입니다. 총 5가지 System call을 추적하여 테스트하고 프로그램을 작성하는 과제이다.

Result

```
os2019202005@ubuntu:~/Downloads/linux-4.19.67$ vi arch/x86/entry/syscalls/syscall_64.tbl
```

```
334     common  rseq                    __x64_sys_rseq
336     common  ftrace                  __X64_sys_ftrace
#
```

위 사진을 통해 ftrace system call을 336번으로 등록한 모습을 확인할 수 있습니다.

```
os2019202005@ubuntu:~/Downloads/linux-4.19.67$ vi include/linux/syscalls.h
```

```
asmlinkage int sys_ftrace(pid_t);
```

include/linux/syscalls.h의 맨 아래 하단에 생성한 시스템 콜 테이블을 등록해줍니다. 어셈블리 코드에서 호출할 수 있도록 등록했습니다.

```
os2019202005@ubuntu:~/Downloads/linux-4.19.67$ mkdir ftrace
os2019202005@ubuntu:~/Downloads/linux-4.19.67$ vi ftrace/ftrace.c
```

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

SYSCALL_DEFINE1(ftrace, pid_t, pid)
{
        printk("pid of ftrace : %d\n",pid);

        return 0;
}
~
```

ftrace라는 새로운 디렉토리를 생성 후 ftrace.c파일을 만들어 System call 함수를

구현했습니다.

```
os2019202005@ubuntu:~/Downloads/linux-4.19.67$ vi ftrace/Makefile
```

```
obj-y := ftrace.o
```

위와 같은 Makefile을 만들어주었습니다.

```
963 core-y          += kernel/ certs/ mm/ fs/ ipc/ security/ crypto/ block/ ftrace/
964
```

core-y 부분에 ftrace를 추가한 모습입니다.

```
os2019202005@ubuntu:~/Downloads/linux-4.19.67$ sudo make
os2019202005@ubuntu:~/Downloads/linux-4.19.67$ sudo make modules_install
os2019202005@ubuntu:~/Downloads/linux-4.19.67$ sudo make install
```

수정된 커널을 컴파일 후 reboot합니다.

```
os2019202005@ubuntu:~/Downloads/linux-4.19.67/test$ vi test.c
```

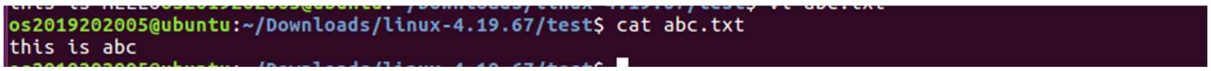
202005@ubuntu: ~/Downloads/linux-4.19.67/test

```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 <= 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;

}
```

주어진 test.c파일을 이용하여 실행했습니다.

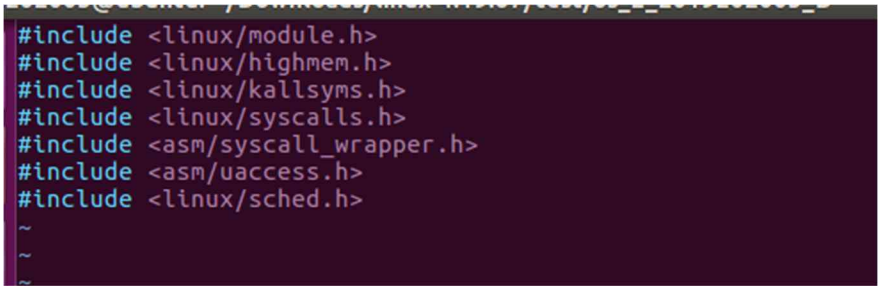
```
os2019202005@ubuntu:~/Downloads/linux-4.19.67/test$ ./test
os2019202005@ubuntu:~/Downloads/linux-4.19.67/test$ dmesg
```

그 후 ftrace.c에 적어둔 내용이 잘 출력되는 모습을 dmesg를 통해 확인할 수 있습니다.



abc.txt파일을 만들고 내용을 확인했습니다.

Ftracehooking.h



위 사진은 ftracehooking.c 및 iotracehooking.c에서 사용하는 header인 ftracehooking.h파일입니다.

#include <linux/module.h> : Linux 커널 모듈을 작성하기 위한 필수 헤더 파일

#include <linux/highmem.h> : highmem영역에 접근하는 데 필요한 헤더 파일

#include <linux/kallsyms.h> : 커널 내부에서 사용되는 symbol 정보에 접근하기 위한 헤더 파일

#include <linux/syscalls.h> : 시스템 콜 관련 정보를 포함하는 헤더 파일

#include <asm/syscall_wrapper.h> : 시스템 콜을 호출하는 데 사용되는 헤더 파일

#include <asm/uaccess.h: 사용자 공간과 커널 간 데이터 복사와 관련된 헤더 파일

#include <linux/sched.h> : 스케줄링과 관련된 헤더 파일

다음으로 ftracehooking.c에 대해 알아보겠습니다.

Ftracehooking.c

```c
#include "ftracehooking.h"

#define __NR_ftrace 336

int open_count = 0;
int read_count = 0;
int close_count = 0;
int write_count = 0;
int lseek_count = 0;

size_t read_bytes = 0;
size_t write_bytes = 0;
char file_name[100] = {0};
```

먼저 헤더 파일들이 포함된 ftracehooking.h파일을 include하고 사용할 변수들을 선언 및 초기화했습니다. 그리고 __NR_ftrace는 Linux 시스템 콜 번호를 나타내는 매크로입니다.

```c
typedef asmlinkage int (*syscall_ptr_t)(const struct pt_regs *);
syscall_ptr_t *syscall_table;
syscall_ptr_t real_ftrace;

EXPORT_SYMBOL(open_count);
EXPORT_SYMBOL(read_count);
EXPORT_SYMBOL(close_count);
EXPORT_SYMBOL(write_count);
EXPORT_SYMBOL(lseek_count);

EXPORT_SYMBOL(read_bytes);
EXPORT_SYMBOL(write_bytes);
EXPORT_SYMBOL(file_name);

pid_t my_pid = 0;
```

typedef asmlinkage int (*syscall_ptr_t)(const struct pt_regs *);를 통해 syscall_ptr_t라는 새로운 데이터 형식을 정의하고 syscall_table과 real_ftrace를 선언했습니다. iotracehooking.c와 연동하기 위해 EXPORT_SYMBOL() 사용했습니다.

```c
static asmlinkage int ftrace(const struct pt_regs *regs)
{
        pid_t pid = regs->di;

        if(pid == 0)
        {
                struct task_struct *curr;
                curr = current;

                printk("[2019202005] %s file[%s] stats [x] read - %d / written - %d\n", curr->comm, file_name, (int)read_bytes, (int)write_bytes);
                printk("open[%d] close[%d] read[%d] write[%d] lseek[%d]\n",open_count,close_count,read_count,write_count,lseek_count);
                printk("OS Assignment 2 ftrace [%d] End\n",my_pid);
        }
        else
        {
                my_pid = pid;
                open_count = 0;
                read_count = 0;
                close_count = 0;
                write_count = 0;
                lseek_count = 0;

                read_bytes = 0;
                write_bytes = 0;
                file_name[0] = '\0';
                printk("OS Assignment 2 ftrace [%d] Start\n",my_pid);
        }
        return 0;
}
```

pid값으로 구조체 내부의 di 레지스터의 값을 할당합니다. 그 후 pid의 값이 0이 아니라면 모든 변수들을 초기화하고 start 문장을 출력합니다. 만약 pid가 0이 아니라면 task_struct *curr을 통해 현재 실행되고 있는 태스크를 출력하고 각 함수들이 호출된 횟수를 출력합니다.

```c
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;
}
asmlinkage int __init hooking_init(void)
{
        syscall_table = (syscall_ptr_t *) kallsyms_lookup_name("sys_call_table");
        make_rw(syscall_table);
        real_ftrace = syscall_table[__NR_ftrace];
        syscall_table[__NR_ftrace] = ftrace;
        return 0;
}

asmlinkage void __exit hooking_exit(void)
{
        make_rw(syscall_table);
        syscall_table[__NR_ftrace] =real_ftrace;
        make_ro(syscall_table);
}
```

1. void make_rw(void *addr)

이는 addr이 속해 있는 페이지의 읽기 및 쓰기 권한을 부여하는 함수입니다. 기본적으로, system call table은 쓰기 권한이 존재하지 않기 때문에 이 함수의 호출을 통해 쓰기 권한이 없는system call table에 쓰기 권한을 부여합니다.

2. void make_ro(void *addr)

addr이 속해 있는 페이지의 읽기 및 쓰기 권한을 회수하는 역할을 합니다.

3.asmlinkage int __init hooking_init(void)

모듈 적재 시 호출되는 함수이며 system call table의 주소를 찾고 쓰기 금지되어 있는 시스템 콜 테이블에 쓰기 권한 부여합니다. 모듈 해제 시 기존의 시스템 콜을 원상 복구하기 위해, 기존 시스템 콜 주소 저장하고 직접 만든 ftrace함수로 대체합니다.

4.static void __exit hooking_exit(void)

모듈 해제 시 호출되는 함수이며 후킹했던 시스템 콜을 원래대로 복원합니다.

Iotracehooking.c

```
#include "ftracehooking.h"

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

typedef asmlinkage long (*syscall_ptr_t)(const struct pt_regs *);
syscall_ptr_t *syscall_table;
```

```
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
```

위 사진을 통해 시스템 콜의 번호를 확인할 수 있습니다. 이를 이용하여 iotracehooking.c에 정의했습니다.

```
extern int open_count;
extern int read_count;
extern int close_count;
extern int write_count;
extern int lseek_count;

extern size_t read_bytes;
extern size_t write_bytes;
extern char file_name[100];

syscall_ptr_t open_real;
syscall_ptr_t read_real;
syscall_ptr_t close_real;
syscall_ptr_t write_real;
syscall_ptr_t lseek_real;
```

Extern을 사용하여 다른 소스 파일에 정의된 변수를 참조할 수 있습니다. 그리고 원래의 시스템 콜 함수를 가리키기 위해 포인터를 선언했습니다.

```
static asmlinkage long ftrace_open(const struct pt_regs *regs)
{
        copy_from_user(file_name, (char*)regs->di, sizeof(file_name));
        open_count++;
        return open_real(regs);
}

static asmlinkage long ftrace_read(const struct pt_regs *regs)
{
        read_bytes += regs->dx;
        read_count++;
        return read_real(regs);
}

static asmlinkage long ftrace_close(const struct pt_regs *regs)
{
        close_count++;
        return close_real(regs);
}

static asmlinkage long ftrace_write(const struct pt_regs *regs)
{
        write_bytes += regs->dx;
        write_count++;
        return write_real(regs);
}

static asmlinkage long ftrace_lseek(const struct pt_regs *regs)
{
        lseek_count++;
        return lseek_real(regs);
}
```

ftrace_open함수에서 copy_from_user(void *to, void *from, unsigned long n)를 사용하여 사용자 영역의 블록 데이터를 커널 영역으로 복사합니다. 함수에서 살펴보면 regs->di에 저장된 data를 fliename에 저장합니다.
그리고 모든 함수들이 몇 번 호출되었는 지 확인하기 위해서 한번 호출될 때마다 count변수를 1씩 증가합니다. 그리고 기존의 함수를 가리키는 포인터를 반환합니다.

ftrace_read함수에서는 read한 데이터의 총 바이트 수를 read_bytes에 저장합니다. 이는 ftrace_write에서도 write입장에서 동일하게 작동합니다.

ftrace_close와 ftrace_lseek함수에서는 함수가 호출된 개수를 저장합니다.

```
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_rw함수와 make_ro함수는 ftracehooking.c에서 설명했으므로 넘어가겠습니다.

```
asmlinkage int __init hooking_init(void)
{
        syscall_table = (syscall_ptr_t *) kallsyms_lookup_name("sys_call_table");
        make_rw(syscall_table);

        open_real = syscall_table[__NR_open];
        read_real = syscall_table[__NR_read];
        close_real = syscall_table[__NR_close];
        write_real = syscall_table[__NR_write];
        lseek_real = syscall_table[__NR_lseek];

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

        return 0;
}
```

```
asmlinkage void __exit hooking_exit(void)
{
        syscall_table = (syscall_ptr_t *) kallsyms_lookup_name("sys_call_table");
        syscall_table[__NR_open] = open_real;
        syscall_table[__NR_read] = read_real;
        syscall_table[__NR_close] = close_real;
        syscall_table[__NR_write] = write_real;
        syscall_table[__NR_lseek] = lseek_real;

        make_ro(syscall_table);
}
```

모듈이 적재되면 hooking_init함수가 호출되며 이 함수는 새로 작성한 5가지의 시스템 콜 함수의 역할을 하게 됩니다. 모듈이 삭제되면 exit_hooking함수가 호출되며 이 함수는 기존의 5가지 시스템 콜 함수로 되돌립니다. 즉, 시스템 콜을 원래대로 복원합니다.

다음으로 open, read, write, lseek, close 시스템콜의 원형을 찾아보겠습니다.

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

```
443 asmlinkage long sys_lseek(unsigned int fd, off_t offset,
444                           unsigned int whence);
445 asmlinkage long sys_read(unsigned int fd, char __user *buf, size_t count);
446 asmlinkage long sys_write(unsigned int fd, const char __user *buf,
447                           size_t count);
```

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

```
1101 SYSCALL_DEFINE3(open, const char __user *, filename, int, flags, umode_t, mode)
1102 {
1103         if (force_o_largefile())
1104                 flags |= O_LARGEFILE;
1105
1106         return do_sys_open(AT_FDCWD, filename, flags, mode);
1107 }
1108
```

```
SYSCALL_DEFINE3(read, unsigned int, fd, char __user *, buf, size_t, count)
{
        return ksys_read(fd, buf, count);
}
```

```
SYSCALL_DEFINE3(write, unsigned int, fd, const char __user *, buf,
                size_t, count)
{
        return ksys_write(fd, buf, count);
}
```

```
SYSCALL_DEFINE1(close, unsigned int, fd)
{
        int retval = __close_fd(current->files, fd);

        /* can't restart close syscall because file table entry was cleared */
        if (unlikely(retval == -ERESTARTSYS ||
                     retval == -ERESTARTNOINTR ||
                     retval == -ERESTARTNOHAND ||
                     retval == -ERESTART_RESTARTBLOCK))
                retval = -EINTR;

        return retval;
}
```

```
SYSCALL_DEFINE3(lseek, unsigned int, fd, off_t, offset, unsigned int, whence)
{
        return ksys_lseek(fd, offset, whence);
}
```

저번 과제에서 배운 cscope를 사용하여 시스템콜 함수들의 원형과 이들의 인자들이 무엇인지에 대해 확인할 수 있습니다.

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

~
```

위 사진은 Makefile이며 ftracehooking.ko 파일과 iotracehooking.ko 파일이 동시에 생성되도록 작성했습니다.

```
os2019202005@ubuntu:~/Downloads/linux-4.19.67/test$ cat abc.txt
this is abc
this is abc
this is HELLOos2019202005@ubuntu:~/Downloads/linux-4.19.67/test$
```

```
os2019202005@ubuntu:~/Downloads/linux-4.19.67/test$ sudo lsmod | grep hooking
os2019202005@ubuntu:~/Downloads/linux-4.19.67/test$ sudo insmod ftracehooking.ko
os2019202005@ubuntu:~/Downloads/linux-4.19.67/test$ sudo insmod iotracehooking.ko
os2019202005@ubuntu:~/Downloads/linux-4.19.67/test$ ./test
os2019202005@ubuntu:~/Downloads/linux-4.19.67/test$ sudo dmesg -c
[ 1173.953403] OS Assignment 2 ftrace [3855] Start
[ 1173.953460] [2019202005] test file[abc.txt] stats [x] read - 20 / written - 26
[ 1173.953478] open[1] close[1] read[4] write[5] lseek[9]
[ 1173.953495] OS Assignment 2 ftrace [3855] End
os2019202005@ubuntu:~/Downloads/linux-4.19.67/test$ sudo rmmod iotracehooking
os2019202005@ubuntu:~/Downloads/linux-4.19.67/test$ sudo rmmod ftracehooking
os2019202005@ubuntu:~/Downloads/linux-4.19.67/test$ ./test
os2019202005@ubuntu:~/Downloads/linux-4.19.67/test$ sudo dmesg -c
[ 1204.658174] ftrace pid:3876
[ 1204.658239] ftrace pid:0
os2019202005@ubuntu:~/Downloads/linux-4.19.67/test$
```

먼저 lsmod명령어를 hooking관련 모듈이 아직 적재되지 않은 것을 확인하고 진행했습니다. Sudo insmod ftracehooking.jo명령어를 통해서 먼저 ftracehooking.ko모듈을 적재하고 sudo insmod iotracehooking명령어를 통해 다음으로 적재했습니다. 그 후 실행파일을 실행시키고 dmesg를 통해 출력 갑을 확인한 결과 과제 요구사항에 맞게 잘 출력되는 모습을 확인할 수 있습니다. 다음은 적재했던 모듈을 삭제하는 과정입니다. Sudo rmmod iotracehookin을 통해 iotracehookin을 먼저 삭제했습니다. 이때 서로 의존성이 있기 때문에 ftracehooking를 먼저 삭제하면 오류가 납니다. 다음으로 sudo rmmod iotracehooking를 통해 삭제 진행 후 실행파일을 실행시킵니다. 다시 한번 dmesg를 통해 출력 값을 확인한 결과 기존의 ftrace.c파일에 적은 내용일 잘 출력되는 모습을 확인할 수 있습니다.

고찰

저번 과제부터 이번 과제까지 커널 컴파일이 정말 오래 걸렸습니다. 그리고 알 수 없는 오류 또한 너무 많이 생겨서 우분투를 계속 새로 설치했던 것 같습니다. 그래도 스냅샷이라는 기능을 알고 난 이후에는 다시 설치하는 일은 없었습니다. 확실히 커널 부분을 건드린 다는 것 자체가 얼마나 신중한 작업인지 알게 되었습니다. 저번 과제보다 이번 과제의 난이도는 확실히 올라갔다고 느꼈으며 생각하는 시간도 훨씬 오래 걸렸습니다. 저번 학기 시스템 프로그래밍 과목 보다는 코드 자체의 양은 훨씬 적지만 너무 생소하고 os에 대한 지식이 아직 부족해서 그랬던 것 같습니다. 그래도 새로 알게 되는 내용들이 많아 계속 신기해하면서 과제를 진행했습니다. 그리고 마지막에 모듈을 적재하고 삭제하는

과정에서 처음에 적재는 제대로 이루어졌는데 삭제하는 과정에서 iotracehooking.ko부터 삭제를 했어야 하는데 ftracehooking.ko부터 삭제를 해서 오류가 났었습니다. 그래도 이 오류는 서로 의존성이 있기 때문에 iotracehooking을 먼저 삭제하고 ftracehooking를 다음에 삭제해야 한다는 점을 찾아 수정할 수 있었습니다.

Reference

강의자료 2023-2_OSLab_04_Systemcall
강의자료 2023-2_OSLab_05_Module_Programming__Wrapping
강의자료 2023-2_OSLab_06_Task_Management