

운영체제실습

assignment 2

담당교수 : 김태석 교수님

학 번 : 2021202058

성 명 : 송채영

1. Introduction

시스템 콜(System Call)에 대해 이해하고 새로운 시스템 콜(fttrace)을 직접 구현해본다. 이를 토대로 커널 모듈(Kernel Module)과 모듈 프로그래밍을 절차를 이해하고 모듈을 추가하고 제거해본다. 또한 Hooking 을 통해 기존의 시스템 콜(System Call)을 대체하는 함수를 구현해보고 동작원리를 이해하고 실습해본다.

2. Result

A. ftrace system call 만들기

우선 step by step 에 따라 system call 번호 336 번에 ftrace system call 을 만든다.

```
os2021202058@ubuntu: ~/Downloads/linux-4.19.67
os2021202058@ubuntu:~$ cd Downloads/
os2021202058@ubuntu:~/Downloads$ cd linux-4.19.67/
os2021202058@ubuntu:~/Downloads/linux-4.19.67$ vi arch/x86/entry/syscalls/syscall_64.tbl
os2021202058@ubuntu:~/Downloads/linux-4.19.67$
```

x86 아키텍처에서 사용되는 64bit system call 에 대한 정보를 포함하는 테이블 파일을 의미한다.

```
#
# 64-bit system call numbers and entry vectors
#
# The format is:
# <number> <abi> <name> <entry point>
#
# The __x64_sys_*() stubs are created on-the-fly for sys_*() system calls
#
# The abi is "common", "64" or "x32" for this file.
#
```

파일 내의 각 라인은 위와 같은 format 을 가진다. Number 은 system call 번호를 의미하며 abi 는 시스템콜을 호출할 때 사용하는 Application Binary Interface 를 의미하며 64 또는 common 으로 설정된다. Name 은 시스템 콜의 이름을, entry point 는 시스템 콜을 처리하는 함수나 코드의 진입 지점을 가리킨다.

```

os2021202058@ubuntu: ~/Downloads/linux-4.19.67
367 526      x32      timer_create      __x32_compat_sys_timer_create
368 527      x32      mq_notify      __x32_compat_sys_mq_notify
369 528      x32      kexec_load      __x32_compat_sys_kexec_load
370 529      x32      waitid      __x32_compat_sys_waitid
371 530      x32      set_robust_list      __x32_compat_sys_set_robust_list
372 531      x32      get_robust_list      __x32_compat_sys_get_robust_list
373 532      x32      vmsplice      __x32_compat_sys_vmsplice
374 533      x32      move_pages      __x32_compat_sys_move_pages
375 534      x32      preadv      __x32_compat_sys_preadv64
376 535      x32      pwritev      __x32_compat_sys_pwritev64
377 536      x32      rt_tsigqueueinfo      __x32_compat_sys_rt_tsigqueueinfo
378 537      x32      recvmmsg      __x32_compat_sys_recvmmsg
379 538      x32      sendmmsg      __x32_compat_sys_sendmmsg
380 539      x32      process_vm_readv      __x32_compat_sys_process_vm_readv
381 540      x32      process_vm_writev      __x32_compat_sys_process_vm_writev
382 541      x32      setsockopt      __x32_compat_sys_setsockopt
383 542      x32      getsockopt      __x32_compat_sys_getsockopt
384 543      x32      io_setup      __x32_compat_sys_io_setup
385 544      x32      io_submit      __x32_compat_sys_io_submit
386 545      x32      execveat      __x32_compat_sys_execveat/ptregs
387 546      x32      preadv2      __x32_compat_sys_preadv64v2
388 547      x32      pwritev2      __x32_compat_sys_pwritev64v2
389 336      common    ftrace      __x64_sys_ftrace
-- INSERT --                                     389,37-57      Bot

```

위 format 에 맞게 ftrace system call 을 system call table 에 등록했다.

```

os2021202058@ubuntu:~/Downloads/linux-4.19.67$ vi include/linux/syscalls.h

```

위 파일은 system call 인터페이스를 정의하고 있다.

```

os2021202058@ubuntu: ~/Downloads/linux-4.19.67
1274      if (force_o_largefile())
1275          flags |= O_LARGEFILE;
1276      return do_sys_open(AT_FDCWD, filename, flags, mode);
1277 }
1278
1279 extern long do_sys_truncate(const char __user *pathname, loff_t length);
1280
1281 static inline long ksys_truncate(const char __user *pathname, loff_t length)
1282 {
1283     return do_sys_truncate(pathname, length);
1284 }
1285
1286 static inline unsigned int ksys_personality(unsigned int personality)
1287 {
1288     unsigned int old = current->personality;
1289
1290     if (personality != 0xffffffff)
1291         set_personality(personality);
1292
1293     return old;
1294 }
1295 asmlinkage int sys_ftrace(pid_t);
1296
-- INSERT --                                     1295,34      99%

```

어셈블리 코드에서 직접 호출할 수 있도록 등록하였다.

```

os2021202058@ubuntu:~/Downloads/linux-4.19.67$ mkdir ftrace
os2021202058@ubuntu:~/Downloads/linux-4.19.67$ vi ftrace/ftrace.c

```

```
os2021202058@ubuntu: ~/Downloads/linux-4.19.67
#include <linux/kernel.h>
#include <linux/syscalls.h>

SYSCALL_DEFINE1(ftrace, pid_t, pid)
{
    printk("SystemCall ftrace's pid is %d.\n", pid);
    return 0;
}

-- INSERT -- 6,50-57 All
```

ftrace system call 을 구현하였다. PID 의 인자를 받는 system call 이므로 ftrace 의 pid 를 출력해하도록 구현을 하였다. 이때 숫자 1 은 해당 시스템 콜이 인자를 1 개 받는다는 것을 의미한다.

```
os2021202058@ubuntu:~/Downloads/linux-4.19.67$ vi ftrace/Makefile
os2021202058@ubuntu:~/Downloads/linux-4.19.67$
```

```
os2021202058@ubuntu: ~/Downloads/linux-4.19.67
obj-y := ftrace.o

-- INSERT -- 1,18 All
```

ftrace system call 을 make 하기 위한 makefile 이다.

```
os2021202058@ubuntu:~/Downloads/linux-4.19.67$ vi Makefile
os2021202058@ubuntu:~/Downloads/linux-4.19.67$
```

```

os2021202058@ubuntu: ~/Downloads/linux-4.19.67
963 core-y      += kernel/ certs/ mm/ fs/ ipc/ security/ crypto/ block/ ftrace/
964
965 vmlinux-dirs := $(patsubst %/,,$(filter %/, $(init-y) $(init-m) \
966                $(core-y) $(core-m) $(drivers-y) $(drivers-m) \
967                $(net-y) $(net-m) $(libs-y) $(libs-m) $(virt-y)))
968
969 vmlinux-alldirs := $(sort $(vmlinux-dirs) $(patsubst %/,,$(filter %/, \
970                $(init-) $(core-) $(drivers-) $(net-) $(libs-) $(virt-))))
971
972 init-y      := $(patsubst %/, %/built-in.a, $(init-y))
973 core-y      := $(patsubst %/, %/built-in.a, $(core-y))
974 drivers-y   := $(patsubst %/, %/built-in.a, $(drivers-y))
975 net-y       := $(patsubst %/, %/built-in.a, $(net-y))
976 libs-y1     := $(patsubst %/, %/lib.a, $(libs-y))
977 libs-y2     := $(patsubst %/, %/built-in.a, $(filter-out %.a, $(libs-y)))
978 virt-y      := $(patsubst %/, %/built-in.a, $(virt-y))
979
980 # Externally visible symbols (used by link-vmlinux.sh)
981 export KBUILD_VMLINUX_INIT := $(head-y) $(init-y)
982 export KBUILD_VMLINUX_MAIN := $(core-y) $(libs-y2) $(drivers-y) $(net-y) $(virt-y)
983 export KBUILD_VMLINUX_LIBS := $(libs-y1)
984 export KBUILD_LDS          := arch/$(SRCARCH)/kernel/vmlinux.lds
985 export LDFLAGS_vmlinux
-- INSERT --
963,72-80 56%

```

Linux kernel 의 Makefile 로 core-y 부분에 함수를 실행할 오브젝트 경로를 수정한 후 재컴파일 & 재부팅을 한다.

```

[ 3305.894763] SystemCall ftrace's pid is 3417.
[ 3305.894782] SystemCall ftrace's pid is 0.

```

과제란에 올라온 test.c 파일을 사용하였으며 컴파일 후 실행했을 때 test.c 파일을 실행하는 process 의 pid 를 출력하는 것을 확인할 수 있다. 이를 통해 ftrace system call 이 system call table 에 잘 등록되었음을 확인했다.

```

os2021202058@ubuntu:~/Downloads/linux-4.19.67/ftracehooking$ cat abc.txt
abcd
os2021202058@ubuntu:~/Downloads/linux-4.19.67/ftracehooking$

```

abc.txt 의 파일 내용이다.

```

#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;
}

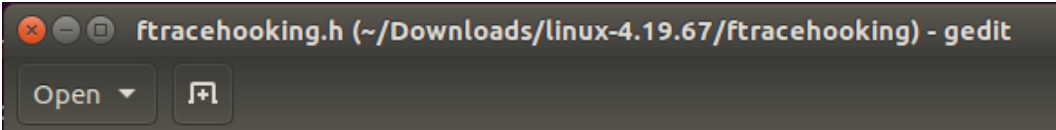
```

주어진 test.c 파일이다.

B. ftrace system call hooking

코드를 구현하는데 필요한 헤더파일을 ftracehooking.h 파일에 저장하였다.

- 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/sched.h>
#include <asm/uaccess.h>
#include <linux/ktime.h>

```

리눅스 커널 모듈을 작성하는데 필요한 헤더파일과 wrapper 하는데 사용되는 헤더파일과 user space 와 kernel space 간 데이터 복사와 관련된 헤더파일, 시간을 출력하기 위해 필요한 헤더파일 등, ftracehooking.c, iotracehooking.c 파일에서 사용하는 헤더파일들을 추가해주었다.

- Ftracehooking.c

```
#include "ftracehooking.h"

#define __NR_ftrace 336

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

int open_count = 0, close_count = 0, read_count = 0, write_count = 0, lseek_count = 0;
char file_name[100] = {0};
size_t read_bytes = 0, written_bytes = 0;

EXPORT_SYMBOL(open_count);
EXPORT_SYMBOL(close_count);
EXPORT_SYMBOL(read_count);
EXPORT_SYMBOL(write_count);
EXPORT_SYMBOL(lseek_count);
EXPORT_SYMBOL(file_name);
EXPORT_SYMBOL(read_bytes);
EXPORT_SYMBOL(written_bytes);

pid_t m_pid = 0;
```

__NR_ftrace - __NR_ftrace 매크로를 ftrace 의 system call 번호인 336 으로 정의하였다.

typedef asmlinkage int (*syscall_ptr_t)(const struct pt_regs *) – ftracehooking.c 부분에서 과제 조건으로 static asmlinkage int ftrace(const struct pt_regs *regs)을 사용해야 했기 때문에 아래 사진과 같이 pt_regs 부분을 참고하여 선언해주었다. 이부분은 커널 내부 system call table 에서 실제 system call 함수를 찾아 호출하는 방식으로 사용된다.

syscall_ptr_t *syscall_table, syscall_ptr_t real_ftrace – system call table 과 기존의 시스템콜을 정의해주었다.

```
#ifdef CONFIG_X86_64
typedef asmlinkage long (*sys_call_ptr_t)(const struct pt_regs *);
#else
typedef asmlinkage long (*sys_call_ptr_t)(unsigned long, unsigned long,
                                         unsigned long, unsigned long,
                                         unsigned long, unsigned long);
#endif /* CONFIG_X86_64 */
extern const sys_call_ptr_t sys_call_table[];
```

Open 횟수, close 횟수, read 횟수, write 횟수, lseek 횟수를 저장할 count 변수와 file name 을 저장할 변수, 읽고 쓴 byte 수를 저장할 read bytes, written bytes 변수와 pid 변수를 정의하고 초기화하였다.

EXPORT_SYMBOL 안에 count 변수들을 넣어줌으로써 다른 모듈에서 사용할 수 있도록 해주었다.

```
static asmlinkage int ftrace(const struct pt_regs *regs)
{
    pid_t pid = regs->di;
    ktime_t start_time, end_time;
    ktime_t elapsed_ktime;
    unsigned long long elapsed_time_ns;

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

        start_time = ktime_get();

        end_time = ktime_get();

        elapsed_ktime = ktime_sub(end_time, start_time);
        elapsed_time_ns = (unsigned long long)ktime_to_ns(elapsed_ktime);

        printk("[2021020508] /%s file[%s] stats [x] read - %d / written - %d\n", current->comm, file_name, (int)read_bytes, (int)written_bytes);
        printk("[%llu ns] open[%d] close[%d] read[%d] write[%d] lseek[%d]\n", elapsed_time_ns, open_count, close_count, read_count, write_count, lseek_count);
        printk("OS Assignment 2 ftrace [%d] End\n", m_pid);
    }
    else
    {
        m_pid = pid;
        open_count = 0;
        close_count = 0;
        read_count = 0;
        write_count = 0;
        lseek_count = 0;
        read_bytes = 0;
        written_bytes = 0;
        file_name[0] = '\0';

        printk(KERN_INFO "OS Assignment 2 ftrace [%d] Start\n", m_pid);
    }

    return 0;
}
```

Syscall table(x86_64, 64bit)의 레지스터 값을 참고하여 코드를 구현하였다. 참고문헌에 링크를 첨부해 두었다.

코드를 살펴보면 다음과 같다.

regs 매개변수에서 PID 값을 추출하여 pid 변수에 저장해주었다. 첫번째 인자에 접근하기 위해 regs->di 를 통해 pid 값을 얻어냈다. Pid 의 값이 0 이라면 trace 종료 부분이므로 과제 출력 양식에 맞게 출력해주었다. Pid 의 값이 0 이 아니라면 trace 시작 부분이므로 실행하는 파일에 대해서만 확인하기 위해 변수를 초기화 해준 후 과제 출력 양식에 맞게 출력해주었다. Ktime 부분은 trace 시작 시간부터 종료까지 걸린 시간을 출력하기 위한 부분이다.

```
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 함수는 읽기 및 쓰기 권한을 부여하는 함수이며 make_ro 는 읽기 및 쓰기 권한을 회수하는 함수이다.

```
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);
}

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

hooking_init 함수는 모듈 적재 시 호출되는 함수이며 sys_call_table 의 ftrace system call을 hooking 해서 새로 구현한 ftrace 함수로 대체한다. hooking_exit 함수는 모듈 해제 시 호출되는 함수로 sys_call_table 의 ftrace system call hooking 을 원상태로 복원한다. 이 부분에서 rmmod 로 iotracehooking 의 권한을 회수한 후 ftracehooking 의 권한을 회수할 때 killed 가 떴다. 읽기 및 쓰기 권한이 없어 생기는 문제라고 생각되어 읽기 및 쓰기 권한을 주는 부분을 추가해주었다.

C. open, read, write, lseek, close 시스템콜의 원형 찾기

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

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

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

cscope-R 을 사용해서 open, read, write, lseek, close 함수의 원형을 찾아보았다. 이를 통해 각 함수의 인자를 확인할 수 있었다.

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

```
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(open, const char __user *, filename, int, flags, umode_t, mode)
{
    if (force_o_largefile())
        flags |= O_LARGEFILE;

    return do_sys_open(AT_FDCWD, filename, flags, mode);
}
```

위의 사진들은 인자의 개수로 검색해본 결과이다.

D. hooking 하여 ftrace_* 함수로 대체

- lotracehooking.c

```
os2021202058@ubuntu: ~/Downloads/linux-4.19.67
#
# 64-bit system call numbers and entry vectors
#
# The format is:
# <number> <abi> <name> <entry point>
#
# The __x64_sys_*() stubs are created on-the-fly for sys_*() system calls
#
# The abi is "common", "64" or "x32" for this file.
#
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" 389L, 15696C      16,1      Top
```

우선 system call 번호를 확인한 후 정의해주었다.

```
#include "ftracehooking.h"

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

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

extern int read_count;
extern int write_count;
extern int open_count;
extern int close_count;
extern int lseek_count;
extern size_t read_bytes;
extern size_t written_bytes;
extern char file_name[100];

syscall_ptr_t real_read;
syscall_ptr_t real_write;
syscall_ptr_t real_open;
syscall_ptr_t real_close;
syscall_ptr_t real_lseek;
```

typedef asmlinkage int (*syscall_ptr_t)(const struct pt_regs *)와 syscall_ptr_t *syscall_table 은 위에서 설명한 것과 같다. Ftracehooking.c 에서 EXPORT_SYMBOL 을 사용해 선언한 변수들을 사용하기 위해 extern 변수를 통해 iotracehooking.c 에서 사용할 수 있게 해주었다. 마지막으로 real_변수는 system call 의 원래 함수를 가리키는 포인터이다.

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

read 함수의 원형에서 필요한 인자는 3 번째, count 이므로 regs->dx 를 통해 얼마나 읽었는지 외부변수 read_bytes 변수에 저장해준 후 read_count 를 증가하고 Hooking 된 system call 의 동작이 기존 system call 과 호환성을 유지하기 위해서 원래의 system call 의 반환값을 반환하였다.

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

write 함수의 원형에서 필요한 인자는 3 번째, count 이므로 regs->dx 를 통해 얼마나 썼는지 외부변수 written_bytes 변수에 저장해준 후 write_count 를 증가하고 Hooking 된 system call 의 동작이 기존 system call 과 호환성을 유지하기 위해서 원래의 system call 의 반환값을 반환하였다.

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

open 함수의 원형에서 필요한 인자는 1 번째, file_name 이므로 regs->di 를 통해 가져왔다. copy_from_user 함수를 사용하여 user space 에서 파일 이름을 kernel space 로 복사하고 open_count 를 증가하고 Hooking 된 system call 의 동작이 기존 system call 과 호환성을 유지하기 위해서 원래의 system call 의 반환값을 반환하였다.

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

close 함수의 원형에서 필요한 인자가 없기 때문에 close_count 를 증가하고 Hooking 된 system call 의 동작이 기존 system call 과 호환성을 유지하기 위해서 원래의 system call 의 반환값을 반환하였다.

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

lseek 함수의 원형에서 필요한 인자가 없기 때문에 lseek_count 를 증가하고 Hooking 된 system call 의 동작이 기존 system call 과 호환성을 유지하기 위해서 원래의 system call 의 반환값을 반환하였다.

```
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;
}
```

Ftracehooking.c 에서와 마찬가지로 make_rw 함수는 읽기 및 쓰기 권한을 부여하는 함수이며 make_ro 는 읽기 및 쓰기 권한을 회수하는 함수이다.

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

    make_rw(syscall_table);

    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_open = syscall_table[__NR_open];
    syscall_table[__NR_open] = ftrace_open;

    real_close = syscall_table[__NR_close];
    syscall_table[__NR_close] = ftrace_close;

    real_lseek = syscall_table[__NR_lseek];
    syscall_table[__NR_lseek] = ftrace_lseek;

    return 0;
}

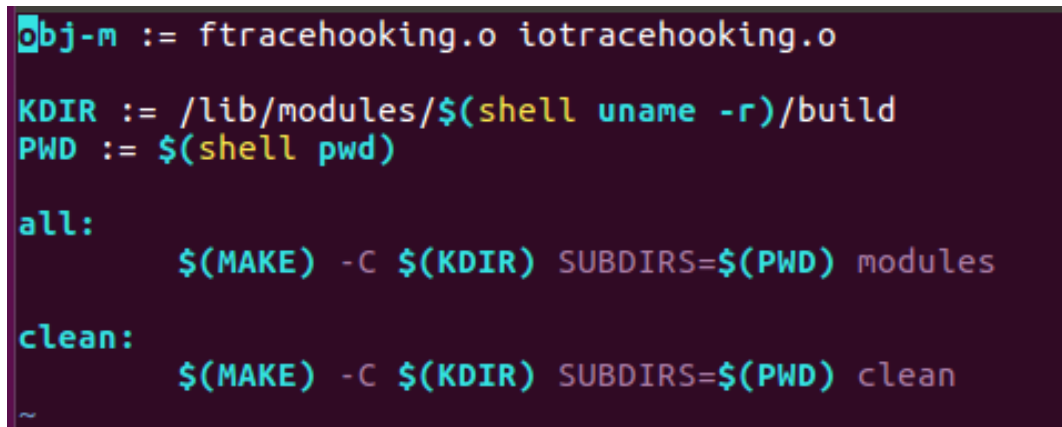
asmlinkage void __exit hooking_exit(void)
{
    syscall_table[__NR_read] = real_read;
    syscall_table[__NR_write] = real_write;
    syscall_table[__NR_open] = real_open;
    syscall_table[__NR_close] = real_close;
    syscall_table[__NR_lseek] = real_lseek;

    make_ro(syscall_table);
}

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

hooking_init 함수는 모듈 적재 시 호출되는 함수이며 sys_call_table 의 ftrace system call을 hooking 해서 새로 구현한 ftrace 함수로 대체한다. hooking_exit 함수는 모듈 해제 시 호출되는 함수로 sys_call_table 의 ftrace system call hooking 을 원상태로 복원한다.

E. 결과화면



```
obj-m := ftracehooking.o 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

~
```

Makefile 이며 ftracehooking 과 iotracehooking 이 동시에 컴파일 되도록 작성하였다.

```
os2021202058@ubuntu:~/Downloads/linux-4.19.67/ftracehooking$ sudo make
[sudo] password for os2021202058:
make -C /lib/modules/4.19.67-2021202058/build SUBDIRS=/home/os2021202058/Downloads/linux-4.19.67/ftracehooking modules
make[1]: Entering directory '/home/os2021202058/Downloads/linux-4.19.67'
  CC [M] /home/os2021202058/Downloads/linux-4.19.67/ftracehooking/iotracehooking.o
  Building modules, stage 2.
  MODPOST 2 modules
  CC /home/os2021202058/Downloads/linux-4.19.67/ftracehooking/iotracehooking.mod.o
  LD [M] /home/os2021202058/Downloads/linux-4.19.67/ftracehooking/iotracehooking.ko
make[1]: Leaving directory '/home/os2021202058/Downloads/linux-4.19.67'
```

```
os2021202058@ubuntu:~/Downloads/linux-4.19.67/ftracehooking$ sudo insmod ftracehooking.ko
os2021202058@ubuntu:~/Downloads/linux-4.19.67/ftracehooking$ sudo insmod iotracehooking.ko
os2021202058@ubuntu:~/Downloads/linux-4.19.67/ftracehooking$ ./test
os2021202058@ubuntu:~/Downloads/linux-4.19.67/ftracehooking$ lsmod | grep hooking
iotracehooking      16384  0
ftracehooking      16384  1 iotracehooking
```

모듈 적재 및 확인하는 과정이다.

```
271.500637] OS Assignment 2 ftrace [2840] Start
271.500664] [2021202058] /test file[abc.txt] stats [x] read - 20 / written - 26
271.500665] [48 ns] open[1] close[1] read[4] write[5] lseek[9]
271.500665] OS Assignment 2 ftrace [2840] End
os2021202058@ubuntu:~/Downloads/linux-4.19.67/ftracehooking$
```

dmesg 로 확인해본 결과이다. trace 시작 커널 메시지, 학번, process_name, file_name, stats, read_bytes, written_bytes, open, close, read, write, lseek 의 횟수, trace 종료 커널 메시지를 요구에 맞게 잘 출력하는 것을 확인할 수 있다.

```
os2021202058@ubuntu:~/Downloads/linux-4.19.67/ftracehooking$ cat abc.txt
abcd
abcd
abcd
abcd
abcd
HELL0os2021202058@ubuntu:~/Downloads/linux-4.19.67/ftracehooking$
```

기존 system call 도 잘 수행하는 것을 확인할 수 있다.

```
os2021202058@ubuntu:~/Downloads/linux-4.19.67/ftracehooking$ sudo rmmod iotracehooking.ko
os2021202058@ubuntu:~/Downloads/linux-4.19.67/ftracehooking$ sudo rmmod ftracehooking.ko
os2021202058@ubuntu:~/Downloads/linux-4.19.67/ftracehooking$ lsmod | grep hooking
os2021202058@ubuntu:~/Downloads/linux-4.19.67/ftracehooking$
```

```
[ 526.109143] SystemCall ftrace's pid is 3340.
[ 526.109185] SystemCall ftrace's pid is 0.
```

모듈 제거 후 lsmod 와 grep 명령어로 확인했을 때 아무것도 뜨지 않는 것을 확인할 수 있으며 기존의 것도 잘 수행되는 것을 확인할 수 있다.

3. 고찰

우선 시스템프로그래밍 과제때와는 다르게 kernel 부분을 다루다 보니 잘못 건들면 오류가 날 것 같아 쉽게 도전하지 못했다. 실제로 ftrace system call 을 system call table 에 등록할 때는 __x64_sys_ftrace 로 쓰고 syscall.h 에서는 asm linkage int ftrace(pid)로 써서 reboot 했을 때 동작하지 않았다. 또한 system call number 가 336 이어서 334 번 사이에 넣어주어야 한다고 생각했지만 이 부분 역시 오류가 났다.

```
#
# x32-specific system call numbers start at 512 to avoid cache impact
# for native 64-bit operation. The __x32_compat_sys stubs are created
# on-the-fly for compat_sys_*() compatibility system calls if X86_X32
# is defined.
#
```

해당 부분은 x32 아키텍처에서의 32bit 와 64bit 의 중간 부분으로 캐시 영향을 최소화하기 위해서 존재하는 부분이다. 그렇기 때문에 새로운 system call 인 ftrace system call 을 맨 아래에 써주어야 하는 이유였다.

또한 가장 어려웠던 부분은 iotracehooking.c 를 구현하는 부분이다. 함수의 원형을 찾고 새로운 함수로 대체하는 부분을 어떻게 구현해야 할지 감이 잘 안 왔던 것 같다. 또한 지금까지 배웠던 것들을 응용할 생각을 해내는 것이 어려웠다. Cscope로 linux system call table 의 레지스터를 어떤 식으로 넘겨주어야 할지 찾아보았는데 결국 찾지 못해 구글링을 통해 알게 되었다. typedef asmlinkage int (*syscall_ptr_t)(const struct pt_regs *) 이부분 역시 pt_regs 의 원형을 찾아볼 생각을 못해 가장 오래 걸렸던 부분인 것 같다. 이번 과제에서는 거의 모든 부분을 cscope 를 활용해 원형을 찾아보고 코드를 구현하는 과정이 많았던 것 같다.

코드를 다 구현하고 보고서를 쓰는 과정에서 실습 자료에 대부분의 개념들이 있었다는 것을 깨달았고 실패하고 도전해보는 과정에서 얻는 것 역시 많았다. 비록 조금의 실수로 인해 reboot 했을 때 검은화면에서 멈춘다는 등 다양한 오류가 있었지만 이로 인해 더 신중하게 생각하고 코딩하고 make 하게 되었다. 또한 이론으로만 배웠던 내용을 직접 실습해보니 이해가 안됐던 부분이 좀 더 쉽게 다가오는 것 같다.

4. Reference

- Linux system call table 정리(32bit, 64bit)

<https://rninche01.tistory.com/entry/Linux-system-call-table-%EC%A0%95%EB%A6%ACx86-x64>

- EXPORT_SYMBOL / EXPORT_SYMBOL_GPL 매크로

<https://blog.naver.com/cre8tor/90193630398>

- 프로그래밍/리눅스 copy_from_user

<https://naito.tistory.com/entry/%ED%94%84%EB%A1%9C%EA%B7%B8%EB%9E%98%EB%B0%8D%EB%A6%AC%EB%88%85%EC%8A%A4-copyfromuser>

- 운영체제실습 강의자료 참조