

운영체제 Assignment 2

이름 : 이준휘

학번 : 2018202046

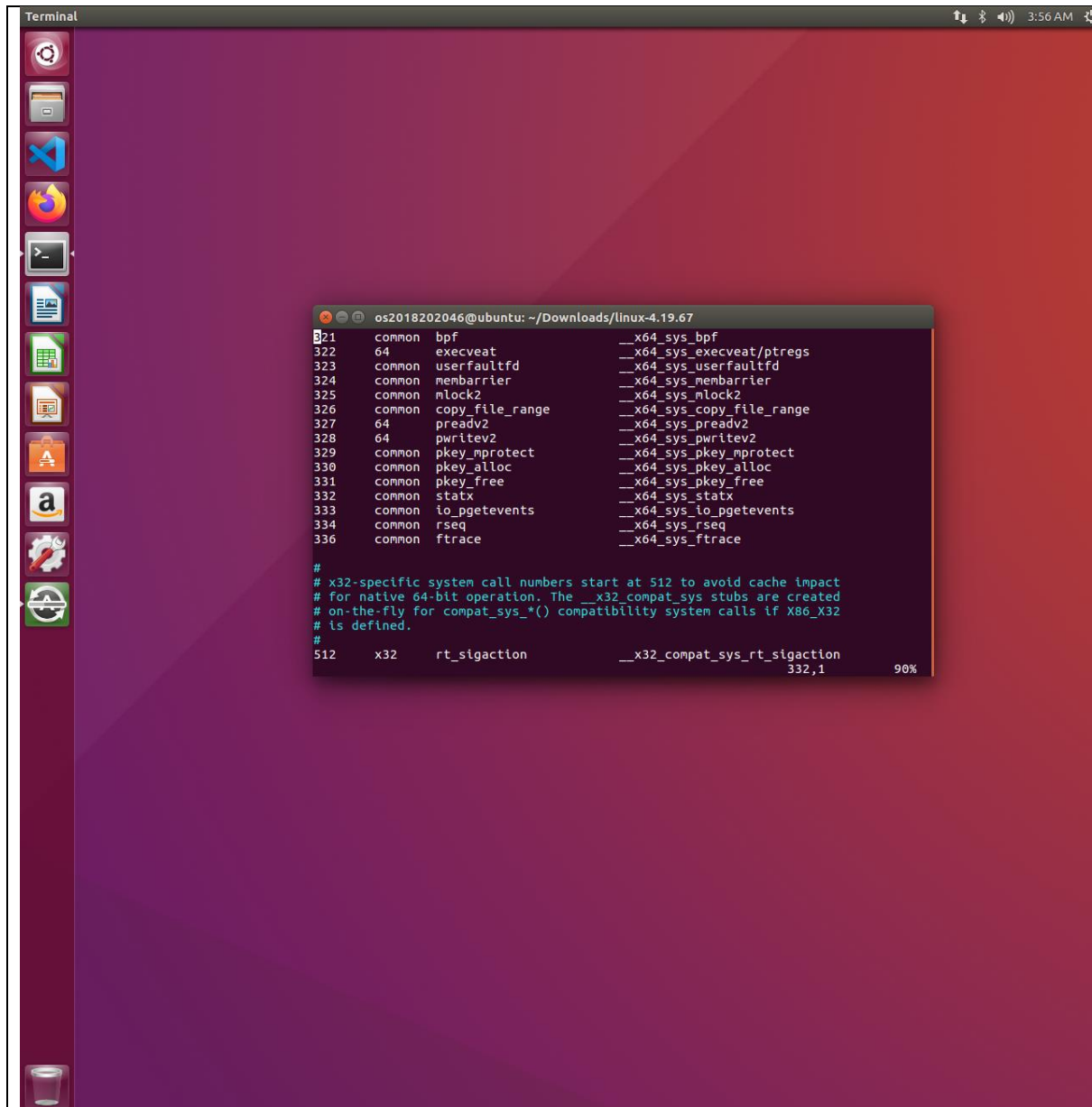
교수 : 최상호 교수님

강의 시간 : 금 1, 2

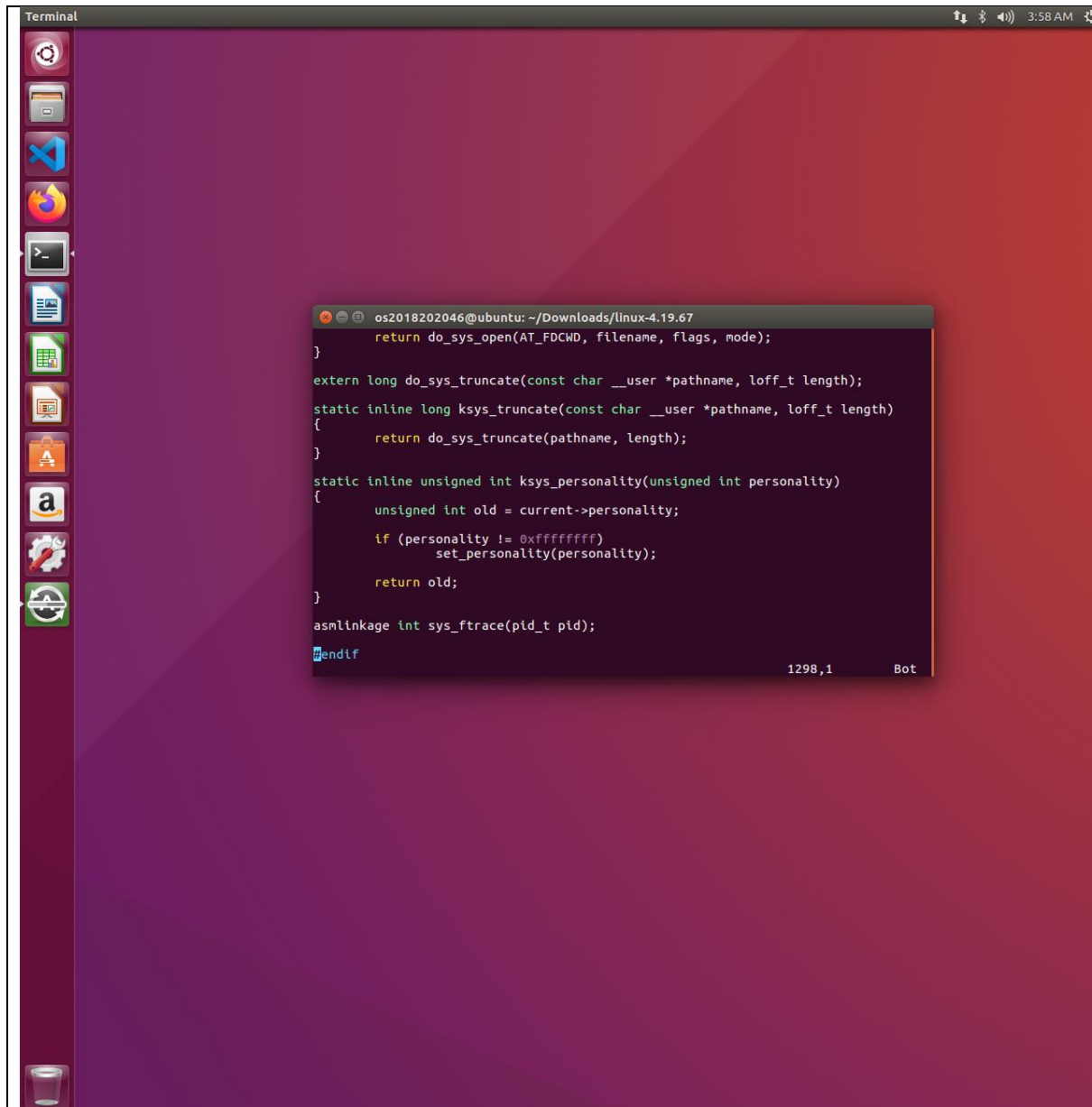
1. Introduction

해당 과제는 두 가지 모듈로 나누어서 테스트를 진행한다. iotracehooking 모듈에서는 open, close, read, write, lseek 모듈을 hijack하여 ftrace_open, ftrace_close, ftrace_read, ftrace_write, ftrace_lseek 모듈로 대체한다. 또한 대체한 모듈에서 얻은 정보를 바탕으로 ftracehooking 모듈에서는 프로세스에서 수행한 syscall의 명령을 tracing한다. 이 때 기존의 system Call 336번에 ftrace를 미리 할당하여 만들어 두며, 이를 hijack하는 방식으로 구현한다. 두 모듈은 하나의 헤더를 공유하며 하나의 Makefile로 동시에 생성된다.

2. Conclusion & Analysis

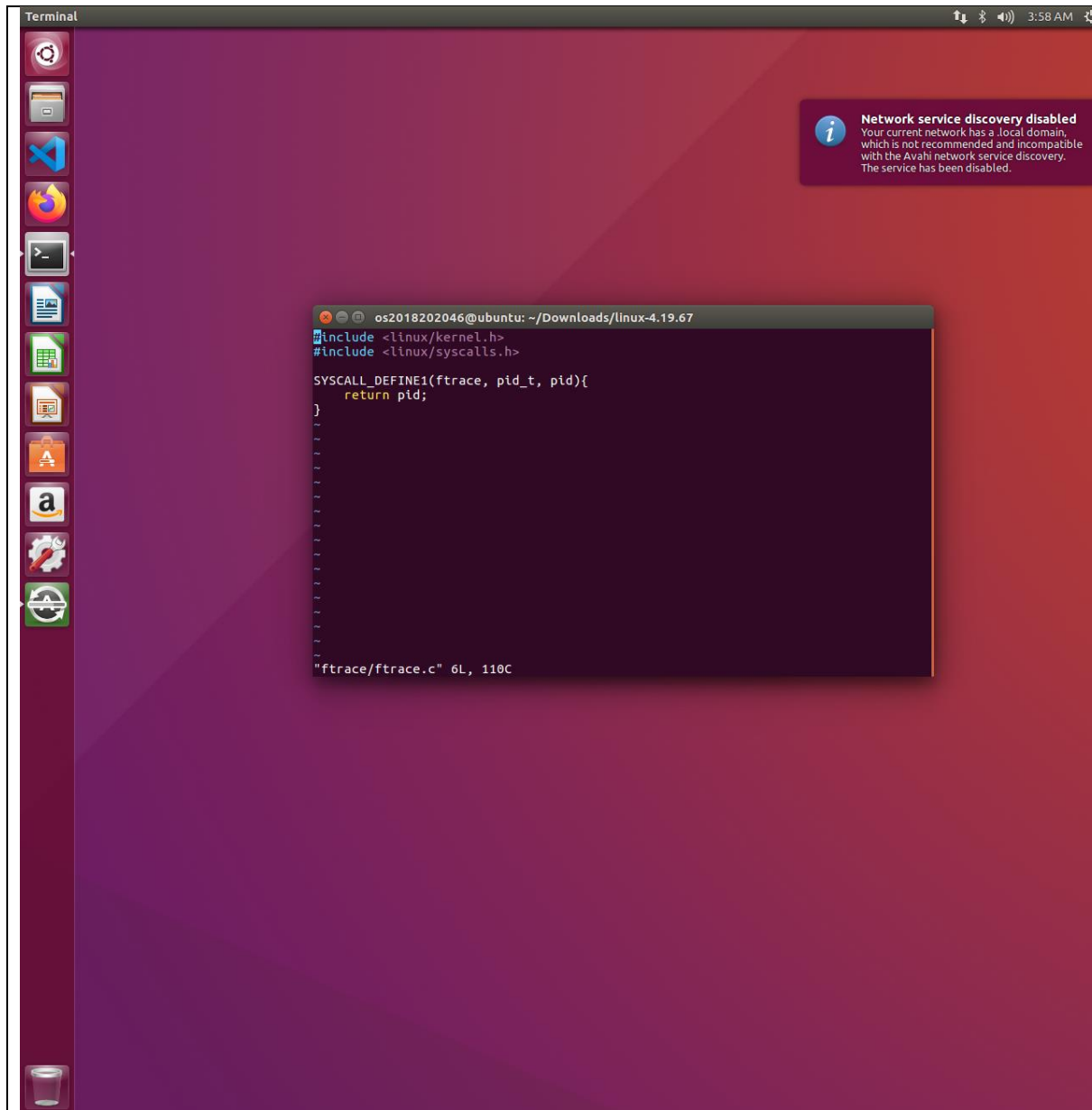


해당 사진은 linux 커널 파일에서 /arch/x86/entry/syscalls/syscall_64.tbl에 저장되어있는 system call table에 ftrace를 추가한 모습이다. Ftrace에 336번을 할당하고 이를 __x64_sys_ftrace함수와 연결시킨다. 해당 프로그램이 window 64-bit에서 수행되기 때문에 syscall_64 파일을 수정하며, 함수명 또한 __x64가 포함되어 있다.

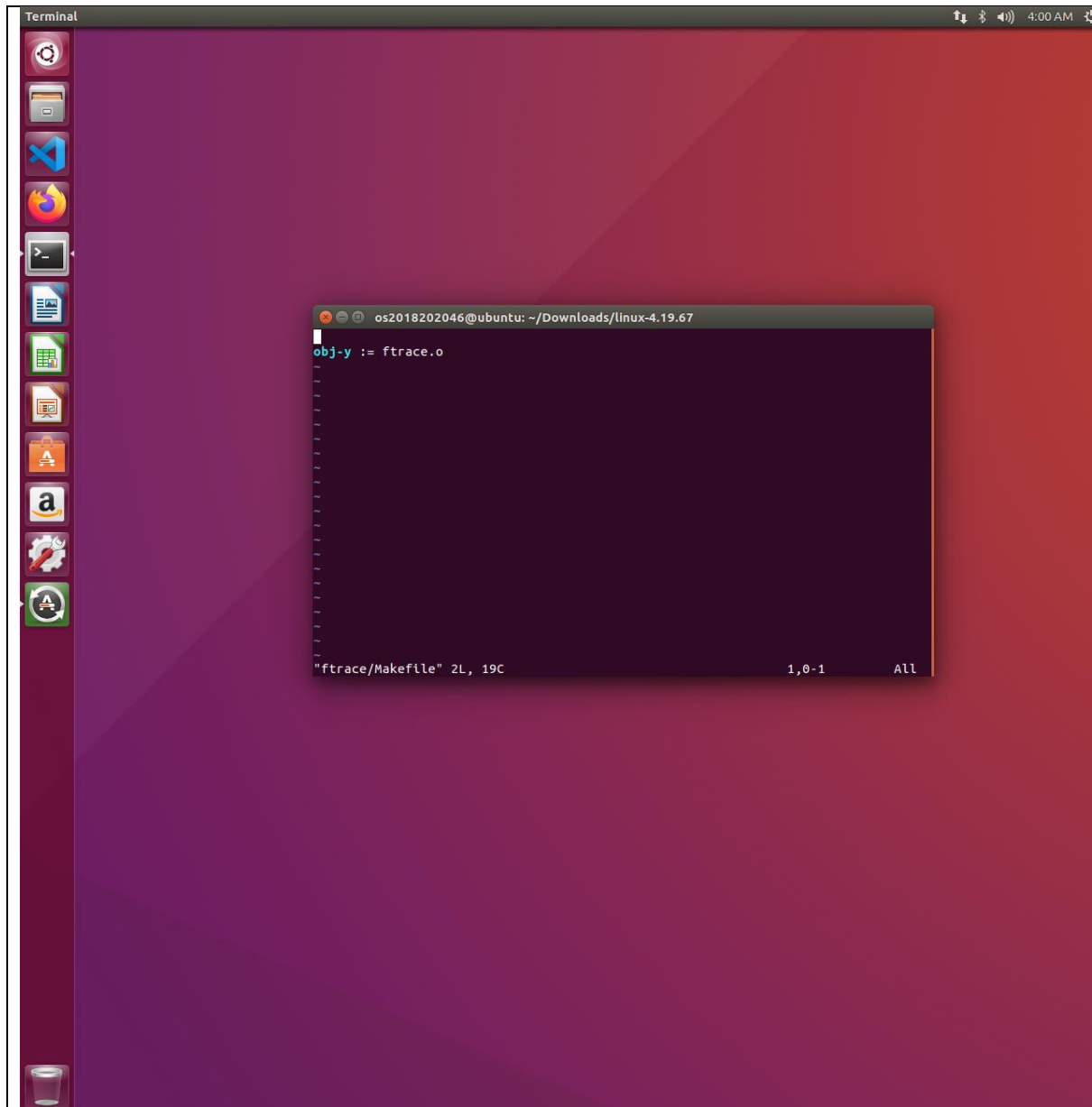


```
Terminal
os2018202046@ubuntu: ~/Downloads/linux-4.19.67
return do_sys_open(AT_FDCWD, filename, flags, mode);
}
extern long do_sys_truncate(const char __user *pathname, loff_t length);
static inline long ksys_truncate(const char __user *pathname, loff_t length)
{
    return do_sys_truncate(pathname, length);
}
static inline unsigned int ksys_personality(unsigned int personality)
{
    unsigned int old = current->personality;
    if (personality != 0xffffffff)
        set_personality(personality);
    return old;
}
asmlinkage int sys_ftrace(pid_t pid);
#endif
1298,1 Bot
```

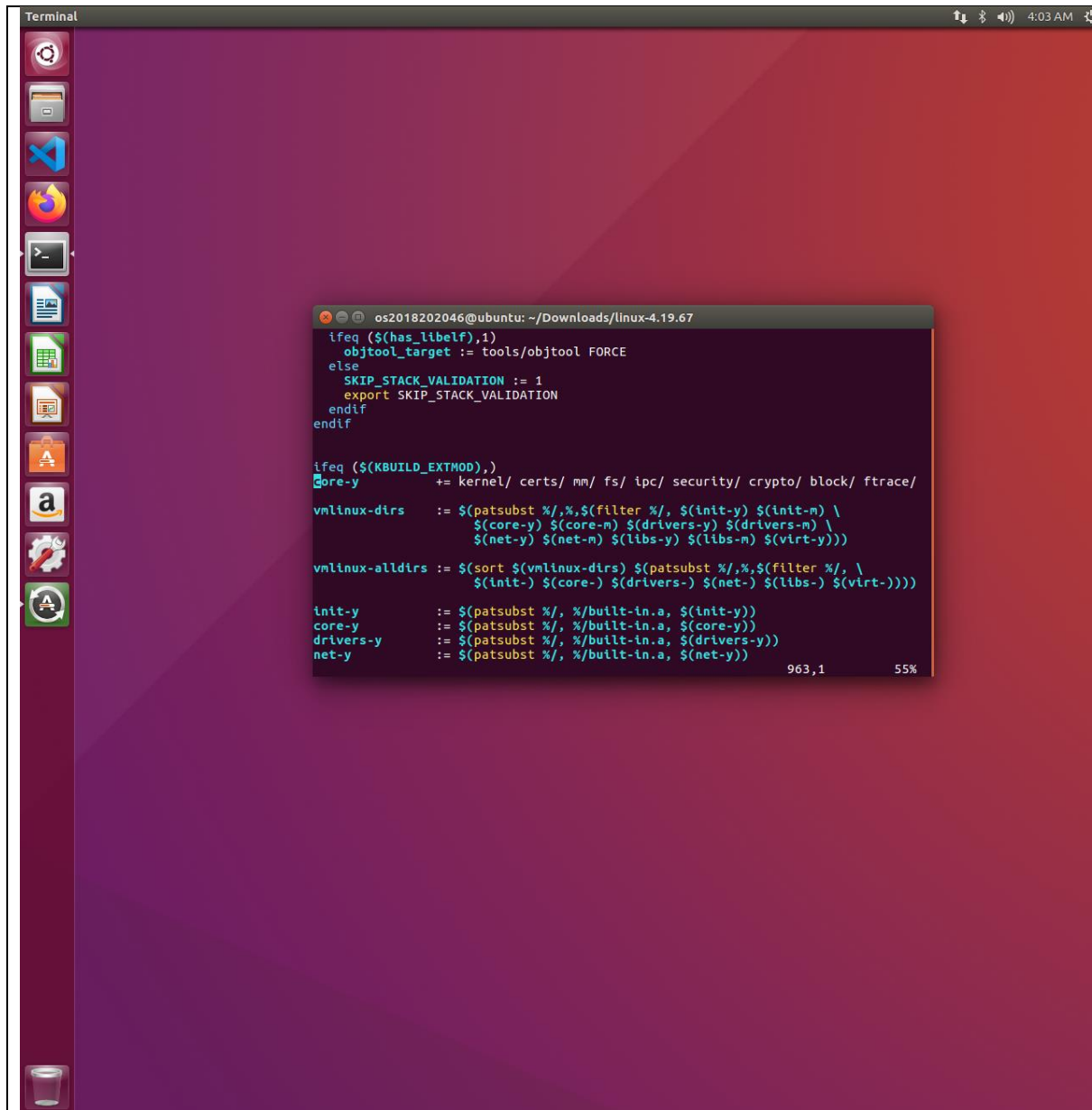
해당 파일 또한 linux 커널 파일에 존재하며, /include/linux/syscalls.h에 저장되어 있다. 해당 파일은 어떠한 system이든 사용이 가능하기 때문에 __x64가 붙지 않으며, 커널에 만들 ftrace 함수를 미리 정의해준다.



해당 파일은 linux kernel Directory에서 새롭게 만든 /ftrace/ftrace.c 파일이다. 해당 파일 내에는 SYSCALL_DEFINE1(ftrace, pid_t, pid)로 함수가 선언되어 있다. 위의 함수는 자동으로 ftrace system call을 만들어주는 매크로로 인자 하나를 받기에 DEFINE1으로 되어있다.



해당 파일은 linux kernel directory에서 ftrace/Makefile로 존재하며, core-y를 통해 해당 파일의 만들 모듈명을 정의해준다.



```
os2018202046@ubuntu: ~/Downloads/linux-4.19.67
ifeq ($(has_libelf),1)
    objtool_target := tools/objtool FORCE
else
    SKIP_STACK_VALIDATION := 1
    export SKIP_STACK_VALIDATION
endif

ifeq ($(KBUILD_EXTMOD),)
core-y      += kernel/ certs/ mm/ fs/ ipc/ security/ crypto/ block/ ftrace/
vmlinux-dirs := $(patsubst %/,%, $(filter %/, $(init-y) $(init-m) \
    $(core-y) $(core-m) $(drivers-y) $(drivers-m) \
    $(net-y) $(net-m) $(libs-y) $(libs-m) $(virt-y)))
vmlinux-alldirs := $(sort $(vmlinux-dirs) $(patsubst %/,%, $(filter %/, \
    $(init-) $(core-) $(drivers-) $(net-) $(libs-) $(virt-))))
init-y      := $(patsubst %/, %/built-in.a, $(init-y))
core-y      := $(patsubst %/, %/built-in.a, $(core-y))
drivers-y   := $(patsubst %/, %/built-in.a, $(drivers-y))
net-y       := $(patsubst %/, %/built-in.a, $(net-y))
963,1      55%
```

다음은 커널의 Makefile 중 core-y의 부분을 찾아 ftrace/를 추가함으로써 해당 모듈을 같이 빌드를 수행하도록 도움을 준다. 위의 과정을 모두 마치고 \$sudo make -j4 -> \$sudo make modules_install -> \$sudo make install 순으로 수행하여 현재 커널에 적용시켰다.


```
os2018202046@ubuntu: ~/os_2_2018202046_B 5:06 AM
#include "ftracehooking.h"
#define __NR_ftrace 336 //System Call Number
asm linkage int (*real_ftrace)(const struct pt_regs *regs); //Save real ftrace Func.
void **syscall_table; //System Call Table
struct ftrace_data ft_data; //Store data(ftrace), extern variable
EXPORT_SYMBOL(ft_data);
//hijack ftrace System Call
static asm linkage int ftrace(const struct pt_regs *regs){
    pid_t pid = real_ftrace(regs); //Get pid from real_ftrace
    //Start Point
    if(pid != 0){
        //Get program name from task
        struct task_struct *findtask = pid_task(find_vpid(pid), PIDTYPE_PID);
        memset((void *)&ft_data, 0, sizeof(ft_data));
        //Store pid and program name
        ft_data.pid = pid;
        ft_data.program_name = findtask->comm;
        printk(KERN_INFO "OS Assignment 2 ftrace [%d] Start\n", pid);
    }
    else{
        //End Point(Print about ft_data)
        printk(KERN_INFO "[2018202046] %s file[%s] start [%d] read - %ld / written - %ld\n",
            ft_data.program_name, ft_data.file_name, ft_data.read_byte, ft_data.write_byte);
        printk(KERN_INFO "open[%d] close[%d] read[%d] write[%d] lseek[%d]\n",
            ft_data.open_count, ft_data.close_count, ft_data.read_count, ft_data.write_count, ft_data.lseek_count);
        printk(KERN_INFO "OS Assignment 2 ftrace [%d] End\n", ft_data.pid);
    }
    return pid;
}
//systemcall table get permission(Read Write)
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;
}
//systemcall table get permission(Read Only)
void make_ro(void *addr){
    unsigned int level;
    pte_t *pte = lookup_address((u64)addr, &level);
    pte->pte = pte->pte &~ _PAGE_RW;
}
//Module initialize Func.
static int __init ftracehooking_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] = ftrace;
    make_ro(syscall_table);
    return 0;
}
//Module exit Func.
static void __exit ftracehooking_exit(void){
    make_rw(syscall_table);
    syscall_table[__NR_ftrace] = real_ftrace;
    make_ro(syscall_table);
}
module_init(ftracehooking_init);
module_exit(ftracehooking_exit);
MODULE_LICENSE("GPL");
```

위의 파일은 ftracehooking.c로 ftrace를 hijacking하는 Function이 정의되어 있다. Ftrace_data 변수는 EXPORT_DEFINE() 함수를 통해 전역 변수로 사용된다. 기존의 System call을 저장할 함수가 존재하며 syscall_table에 kallsyms_lookup_name("sys_call_table")의 값을 대입하여 system call의 table을 가져와 이를 ftrace system call을 hijacking한다. 이 때 table에는 쓰기 권한이 없으므로 임시로 쓰기 권한을 부여한 뒤 바꾼 후 권한을 반환한다. Hijack을 수행하는 함수에서는 register들의 값인 struct pt_regs *reg를 인자로 받는다. 우선 기존 함수를 그대로 수행하여 pid 값을 받은 후 해당 pid값에 따라 다음 명령을 수행한다. 0이 아닐 경우 해당 pid에 대한 ftrace를 수행하기 위해 메모리를 초기화 하고 task_struct에서 pid 값을 통해 현재에 해당하는 task_struct를 찾아 해당 구조체에서 이름을 가져온다. 만약 pid가 0일 경우에는 기존에 ftrace를 수행한 모든 데이터를 출력한다.

```
#include "ftracehooking.h"

//System call hook
#define __NR_OPEN 2
#define __NR_CLOSE 3
#define __NR_READ 0
#define __NR_WRITE 1
#define __NR_LSEEK 8

//Original Func
asm(
    long (*real_open)(const struct pt_regs *regs);
    long (*real_close)(const struct pt_regs *regs);
    long (*real_read)(const struct pt_regs *regs);
    long (*real_write)(const struct pt_regs *regs);
    long (*real_lseek)(const struct pt_regs *regs);
);

void __syscall_table; //System call table
extern struct ftrace_data ft_data; //Store at ftrace's data

//Hijack open's System call
static asm(
    long ftrace_open(const struct pt_regs *regs){
        copy_from_user(&ft_data.file_name, (const char *)regs->di, sizeof(ft_data.file_name)); //Copy user space's file name data
        kernel_open(ft_data.open_count++);
        return real_open(regs);
    }
);

//Hijack close's System call
static asm(
    long ftrace_close(const struct pt_regs *regs){
        ft_data.close_count++;
        return real_close(regs);
    }
);

//Hijack read's System call
static asm(
    long ftrace_read(const struct pt_regs *regs){
        ssize_t size = real_read(regs);
        ft_data.read_count++;
        ft_data.read_byte += size;
        return size;
    }
);

//Hijack write's System call
static asm(
    long ftrace_write(const struct pt_regs *regs){
        ssize_t size = real_write(regs);
        ft_data.write_count++;
        ft_data.write_byte += size;
        return size;
    }
);

//Hijack lseek's System call
static asm(
    long ftrace_lseek(const struct pt_regs *regs){
        ft_data.lseek_count++;
        return real_lseek(regs);
    }
);

//Systemcall table get permission(Read Write)
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;
}

//Systemcall table get permission(Read Only)
void make_ro(void *addr){
    unsigned int level;
    pte_t *pte = lookup_address((u64)addr, &level);
    pte->pte = pte->pte & ~_PAGE_RW;
}

//Module initialize Func
static int __init iotracehooking_init(void){
    syscall_table = (void**)kaliym_lookup_name("sys_call_table");
    make_rw(syscall_table);
    real_open = syscall_table[__NR_OPEN];
    real_close = syscall_table[__NR_CLOSE];
    real_read = syscall_table[__NR_READ];
    real_write = syscall_table[__NR_WRITE];
    real_lseek = syscall_table[__NR_LSEEK];

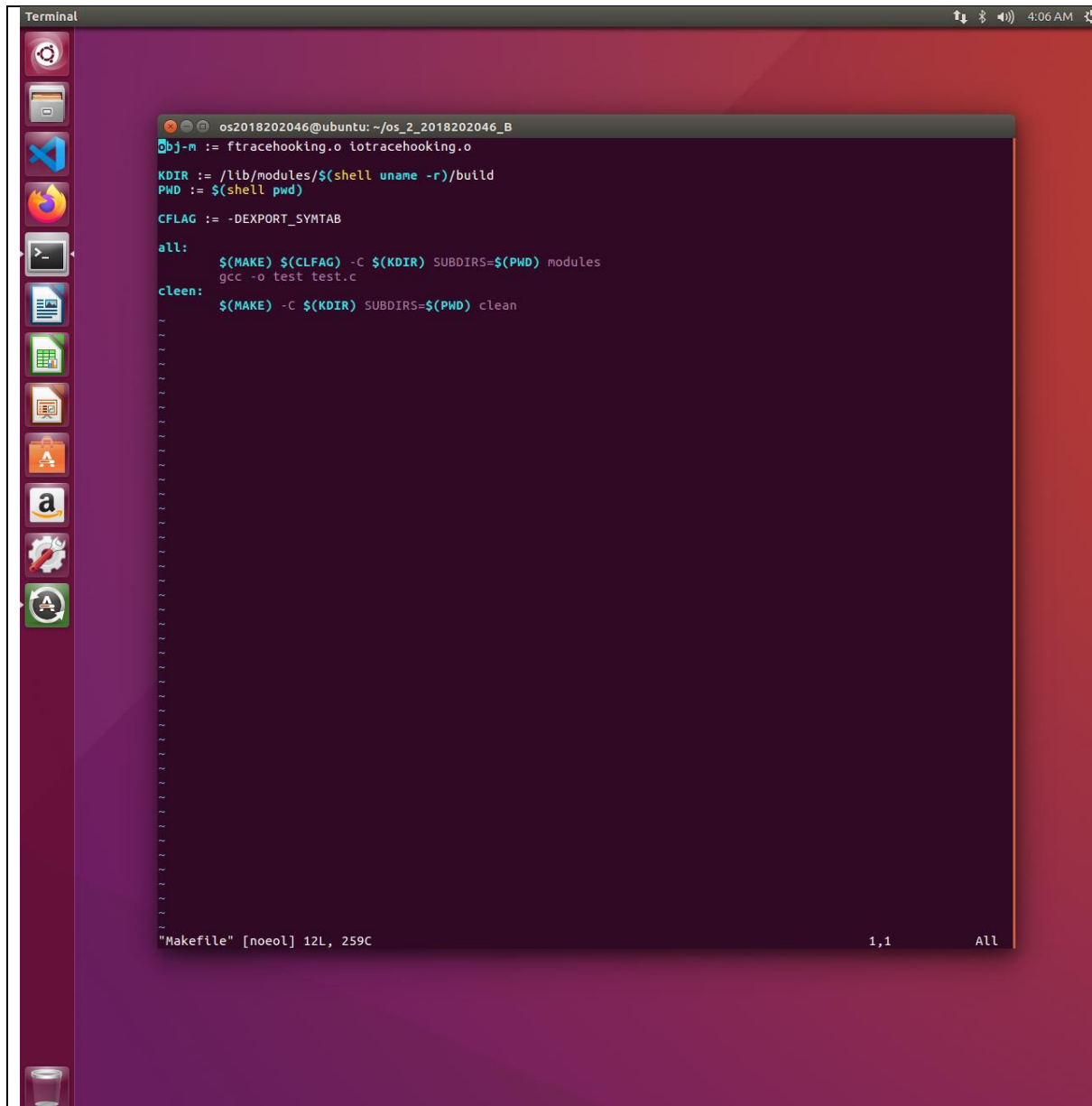
    syscall_table[__NR_OPEN] = ftrace_open;
    syscall_table[__NR_CLOSE] = ftrace_close;
    syscall_table[__NR_READ] = ftrace_read;
    syscall_table[__NR_WRITE] = ftrace_write;
    syscall_table[__NR_LSEEK] = ftrace_lseek;
    make_ro(syscall_table);

    return 0;
}

//Module exit Func
static void __exit iotracehooking_exit(void){
    make_rw(syscall_table);
    syscall_table[__NR_OPEN] = (void *)real_open;
    syscall_table[__NR_CLOSE] = (void *)real_close;
    syscall_table[__NR_READ] = (void *)real_read;
    syscall_table[__NR_WRITE] = (void *)real_write;
    syscall_table[__NR_LSEEK] = (void *)real_lseek;
    make_ro(syscall_table);
}

module_init(iotracehooking_init);
module_exit(iotracehooking_exit);
MODULE_LICENSE("GPL");
```

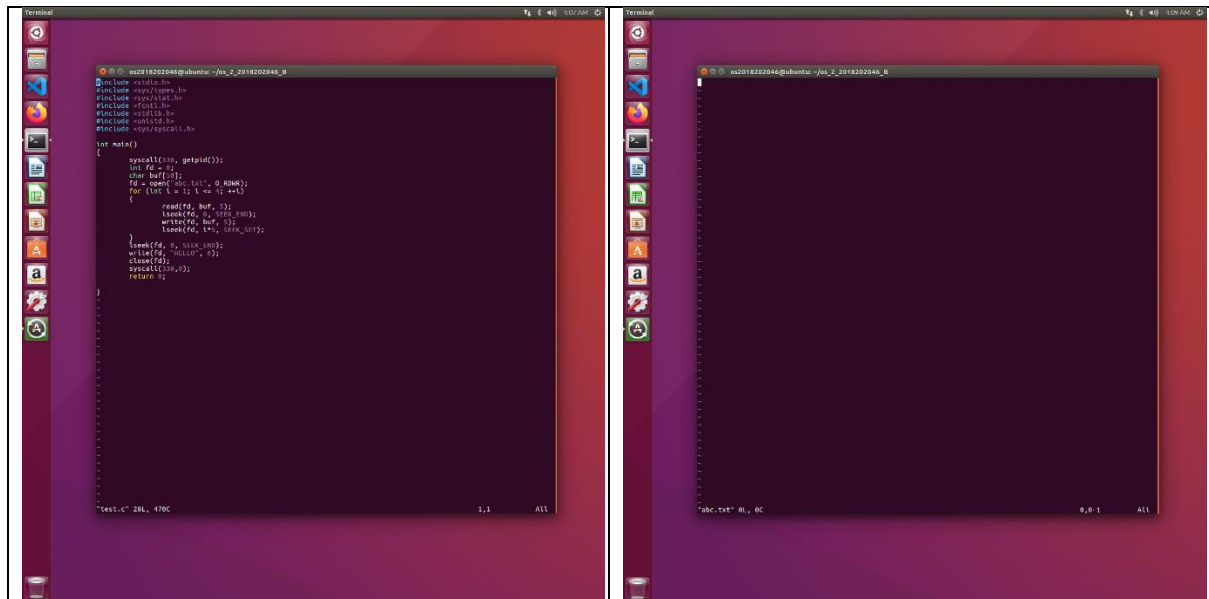
해당 파일은 iotracehooking.c 파일을 나타낸다. 해당 파일도 이전과 마찬가지로 System Call 번호와 기존 함수를 저장할 함수, syscall_table이 선언되어 있으며, ftracehooking.c에서 선언한 전역 변수 extern struct ftrace_data ft_data가 존재한다. 모듈에서도 비슷하게 syscall_table을 조작하게 된다. Open을 hijack하는 함수에서는 copy_from_user를 통해 user space의 파일명 데이터에 해당하는 레지스터인 regs->di를 kernel space로 복사해온다. Read/Write를 hijack하는 함수에서는 ssize_t size에 기존 system call 함수를 수행한 결과인 크기를 받아 이를 xx_byte에 늘려주고 반환하는 형식을 취한다. 이외의 함수도 마찬가지로 기존의 system call을 거의 그대로 사용하며, 반환 전에 해당하는 count를 늘려준다.



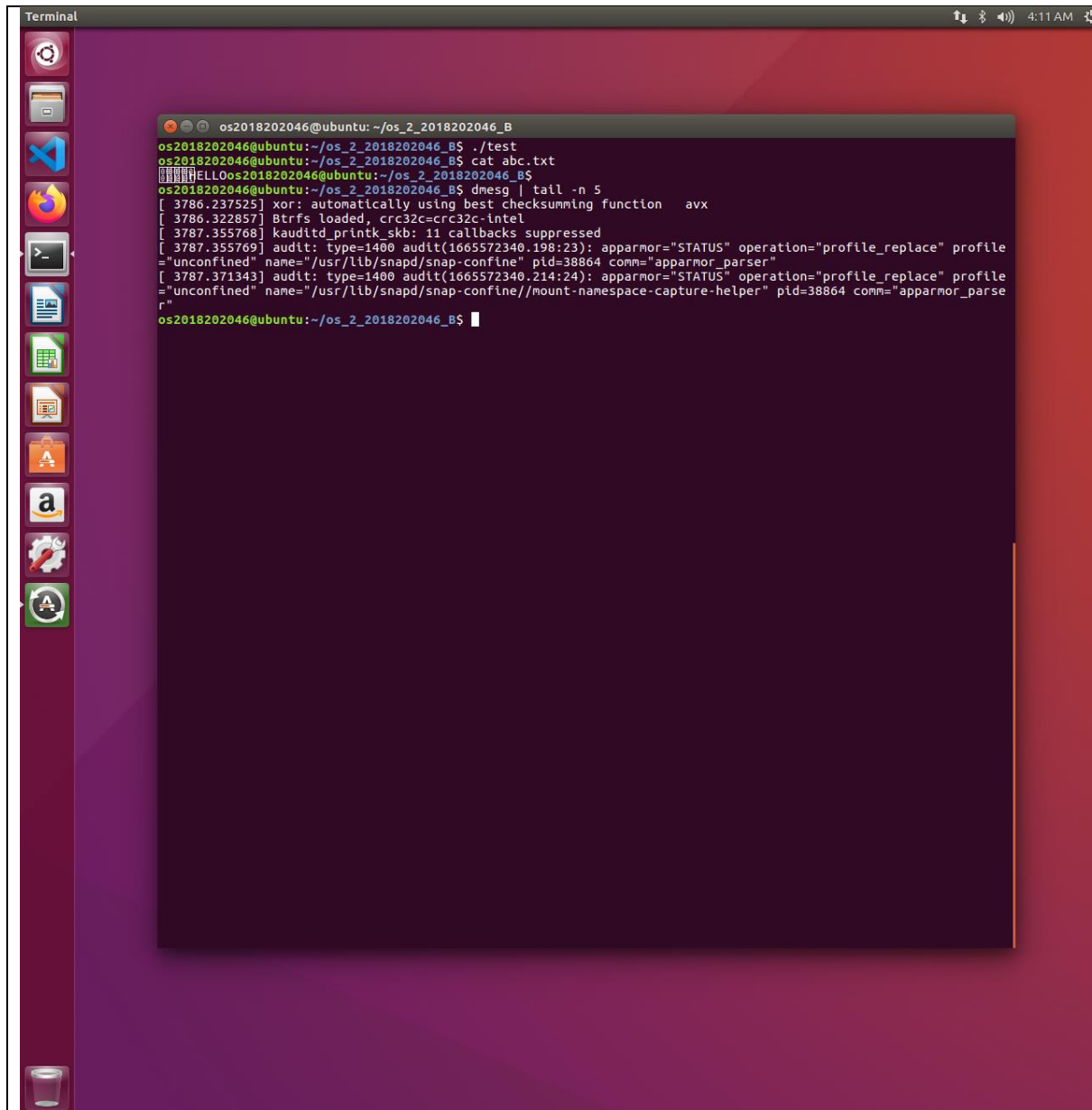
```
Terminal
os2018202046@ubuntu: ~/os_2_2018202046_B
Obj-m := ftracehooking.o iotracehooking.o
KDIR := /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)
CFLAG := -DEXPORT_SYMTAB
all:
    $(MAKE) $(CFLAG) -C $(KDIR) SUBDIRS=$(PWD) modules
    gcc -o test test.c
clean:
    $(MAKE) -C $(KDIR) SUBDIRS=$(PWD) clean

"Makefile" [noel] 12L, 259C
1,1 All
```

해당 파일은 ftracehooking 모듈과 iotracehooking 모듈을 한 번에 만들어주는 모듈이다. Obj-m 에는 만들 모듈들의 이름이 들어가며, KDIR에는 module builder의 위치, PWD에는 현재 경로가 들어간다. CFLAG에는 EXPORT_DEFINE을 사용할 시 필요한 flag인 -DEXPORT_SYMTAB이 들어가 게 된다. 해당 파일에서는 과제에서 준 test.c도 같이 컴파일 되도록 임시로 만들었다.



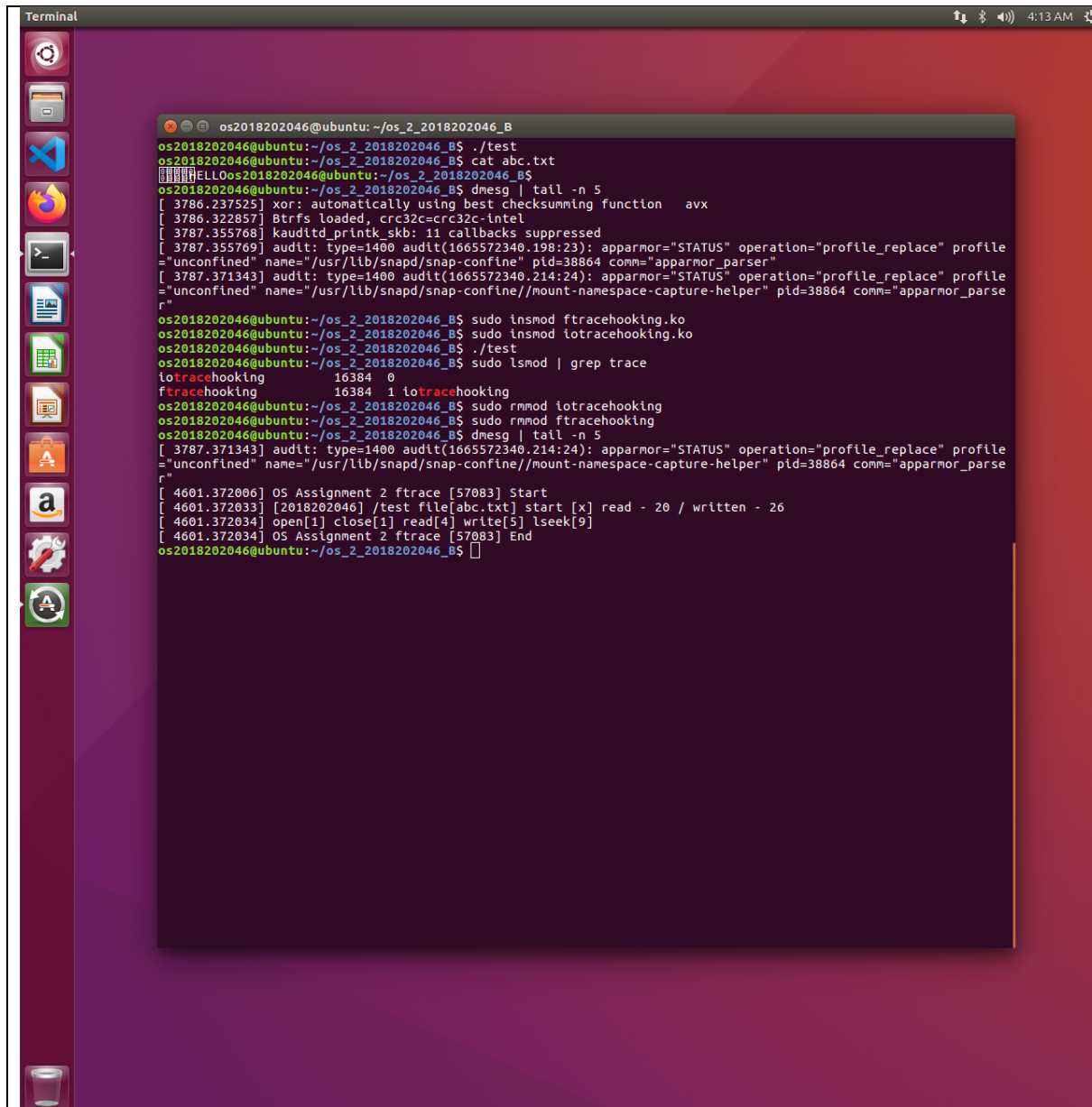
해당 파일은 기존에 주어진 파일인 test.c 파일과 해당 파일에서 사용하는 abc.txt 파일이 다음과 같이 구성되어 있음을 보인다. 해당 test.c에서는 ftrace를 활성화시킨 후, open을 통해 abc.txt를 연다. 이후 4번의 반복동안 read 5byte 1회, write 5byte 1회, lseek 2회를 수행한다. 반복문을 탈출한 후 lseek 1회와 write 6byte 1회를 추가로 수행한 후 close를 수행한다. 그 후 trace를 종료한다. 위의 결과를 예상했을 때 open 1회, close 1회, read 20byte 4회, write 26byte 5회, lseek 9회를 수행해야 한다.



The screenshot shows a terminal window titled "Terminal" with a dark background and a light-colored text area. The terminal displays the following commands and output:

```
os2018202046@ubuntu: ~/os_2_2018202046_B
os2018202046@ubuntu:~/os_2_2018202046_B$ ./test
os2018202046@ubuntu:~/os_2_2018202046_B$ cat abc.txt
HELLOos2018202046@ubuntu:~/os_2_2018202046_B$
os2018202046@ubuntu:~/os_2_2018202046_B$ dmesg | tail -n 5
[ 3786.237525] xor: automatically using best checksumming function   avx
[ 3786.322857] Btrfs loaded, crc32c=crc32c-intel
[ 3787.355768] kauditd_printk_skb: 11 callbacks suppressed
[ 3787.355769] audit: type=1400 audit(1665572340.198:23): apparmor="STATUS" operation="profile_replace" profile
="unconfined" name="/usr/lib/snapd/snap-confine" pid=38864 comm="apparmor_parser"
[ 3787.371343] audit: type=1400 audit(1665572340.214:24): apparmor="STATUS" operation="profile_replace" profile
="unconfined" name="/usr/lib/snapd/snap-confine//mount-namespace-capture-helper" pid=38864 comm="apparmor_parse
r"
os2018202046@ubuntu:~/os_2_2018202046_B$
```

위의 동작은 기존의 system call을 사용하여 ftrace를 진행하지 않고 파일을 수행시킨 결과를 보인다. Cat을 통해 abc.txt를 출력하였을 때 다음과 같은 결과를 보이고 dmesg에는 아무 결과가 찍히지 않는다.

A terminal window titled 'Terminal' with a dark background and a vertical sidebar of application icons on the left. The terminal shows a series of commands and their outputs. The user is logged in as 'os2018202046' on an 'ubuntu' machine, with the current directory being '~/os_2_2018202046_B'. The commands executed are: 1. './test' which produces some garbled output. 2. 'cat abc.txt' which outputs 'ELLO'. 3. 'dmesg | tail -n 5' which shows kernel messages including 'xor: automatically using best checksumming function avx', 'Btrfs loaded, crc32c=crc32c-intel', and 'kauditd_printk_skb: 11 callbacks suppressed'. 4. 'sudo insmod ftracehooking.ko' and 'sudo insmod iotracehooking.ko'. 5. 'sudo rmmod iotracehooking' and 'sudo rmmod ftracehooking'. 6. 'dmesg | tail -n 5' which shows audit messages from 'apparmor_parser' and 'mount-namespcapture-helper'. 7. 'lsmod | grep trace' which shows 'iotracehooking' and 'ftracehooking' loaded. 8. 'rmmod iotracehooking' and 'rmmod ftracehooking'. 9. 'dmesg | tail -n 5' which shows 'OS Assignment 2 ftrace [57083] Start', file operations on '/test file[abc.txt]', and 'OS Assignment 2 ftrace [57083] End'.

```
os2018202046@ubuntu: ~/os_2_2018202046_B
os2018202046@ubuntu:~/os_2_2018202046_B$ ./test
os2018202046@ubuntu:~/os_2_2018202046_B$ cat abc.txt
ELLOos2018202046@ubuntu:~/os_2_2018202046_B$
os2018202046@ubuntu:~/os_2_2018202046_B$ dmesg | tail -n 5
[ 3786.237525] xor: automatically using best checksumming function  avx
[ 3786.322857] Btrfs loaded, crc32c=crc32c-intel
[ 3787.355768] kauditd_printk_skb: 11 callbacks suppressed
[ 3787.355769] audit: type=1400 audit(1665572340.198:23): apparmor="STATUS" operation="profile_replace" profile
="unconfined" name="/usr/lib/snapd/snap-confine" pid=38864 comm="apparmor_parser"
[ 3787.371343] audit: type=1400 audit(1665572340.214:24): apparmor="STATUS" operation="profile_replace" profile
="unconfined" name="/usr/lib/snapd/snap-confine/mount-namespcapture-helper" pid=38864 comm="apparmor_parse
r"
os2018202046@ubuntu:~/os_2_2018202046_B$ sudo insmod ftracehooking.ko
os2018202046@ubuntu:~/os_2_2018202046_B$ sudo insmod iotracehooking.ko
os2018202046@ubuntu:~/os_2_2018202046_B$ ./test
os2018202046@ubuntu:~/os_2_2018202046_B$ sudo lsmod | grep trace
iotracehooking          16384  0
ftracehooking          16384  1 iotracehooking
os2018202046@ubuntu:~/os_2_2018202046_B$ sudo rmmod iotracehooking
os2018202046@ubuntu:~/os_2_2018202046_B$ sudo rmmod ftracehooking
os2018202046@ubuntu:~/os_2_2018202046_B$ dmesg | tail -n 5
[ 3787.371343] audit: type=1400 audit(1665572340.214:24): apparmor="STATUS" operation="profile_replace" profile
="unconfined" name="/usr/lib/snapd/snap-confine/mount-namespcapture-helper" pid=38864 comm="apparmor_parse
r"
[ 4601.372006] OS Assignment 2 ftrace [57083] Start
[ 4601.372033] [2018202046] /test file[abc.txt] start [x] read - 20 / written - 26
[ 4601.372034] open[1] close[1] read[4] write[5] lseek[9]
[ 4601.372034] OS Assignment 2 ftrace [57083] End
os2018202046@ubuntu:~/os_2_2018202046_B$
```

위의 결과는 모듈을 적용하여 ./test 파일을 수행한 후 결과를 보인다. 해당 결과에서 dmesg의 마지막에 기존에 없던 출력이 생겼으며, 이 곳에 실행 파일명, 연 파일명, 읽기&쓰기 byte 수, 각 함수 사용 횟수가 기록되어 있다. 해당 결과가 test.c의 예상 결과와 일치하는 것을 보인다. 이를 통해 해당 모듈이 정상적으로 구현되었음을 알 수 있다.

3. Consideration

해당 과제를 통해 Linux에서 System call이 어떠한 과정을 거치며 호출되는지 순차적으로

알 수 있었다. 처음 접근할 때 ftrace hooking을 수행할 때 SYSCALL_DEFINEx 매크로를 사용하였다가 해당 작업은 매크로를 생성하는 것이 아니라 매크로를 가져오는 것이기에 부적절하다는 것을 알게 되었다. 또한 파일에서의 해당 함수를 찾아보았을 때와 다르게 실제로는 함수로의 직접 접근으로 인한 오류를 방지하기 위해 struct pt_reg* 인자를 통해 값을 전달한 후 해당 함수가 다시 실제 함수로 mapping된다는 것을 알 수 있었다. 또한 해당 pt_reg에는 각 system call마다 사용하는 reg의 위치가 다르다는 것을 알 수 있었다. 마지막으로 전역 변수를 사용하기 위해 EXPORT_DEFINE() 사용해야 한다는 사실을 공부할 수 있던 과제였다.

4. Reference

- 강의자료만을 참고