운영체제실습

assignment 2

학　　과: 컴퓨터정보공학부

담당교수: 김태석 교수님

학　　번: 2019202050

성　　명: 이강현

1. Introduction

System call 이 커널 내에서 어떻게 동작하는지에 대한 이해를 기반으로 직접 system call 을 만들고 등록하여 사용해본다. System call 을 hijacking 하여 이를 추적할 수 있는 모듈을 만들고 모듈을 적용해보면서 개념을 학습한다. 추가적으로 tast_struct 구조체에 대해 학습하면서 process descriptor 가 무엇인지 어떠한 정보들이 존재하고 이를 어떻게 활용할 수 있는지 알아본다.

2. Result

1 차적으로 open,close,read,write,lseek 는 이미 system call 이 존재하므로 Hooking 이 가능하지만 ftrace 라는 system call 은 존재하지 않기에 만들어 주어야 한다.

System call table 을 확인하자.

```
os2019202050@ubuntu: ~/Downloads/linux-4.19.67
os2019202050@ubuntu:~/Downloads/linux-4.19.67$ vi arch/x86/entry/syscalls/syscall_64.tbl
```

```
546     x32     preadv2                 __x32_compat_sys_preadv64v2
547     x32     pwritev2                __x32_compat_sys_pwritev64v2
336     common  ftrace                  __x64_sys_ftrace
                                                                389,1         Bot
```

```
# 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.
```

과제 목적에 맞게 ftrace 라는 system call 을 system call table 에 336 번으로 등록하였다.

등록 방식은 주석을 참고하였고 이름은 ftrace, entry point 는 x64 system call 로 entry point 는 추후 hooking 과정에서 system call 에서 인자를 어떻게 사용하는지에 대한 중요한 단서가 된다.

```
os2019202050@ubuntu:~/Downloads/linux-4.19.67$ vi include/linux/syscalls.h
```

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

System call 헤더파일에 table 에 등록한 ftrace 를 선언해두고

os2019202050@ubuntu: ~/Downloads/linux-4.19.67/ftrace

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

SYSCALL_DEFINE1(ftrace, pid_t, pid)
{
        printk("ftrace is called by pid:%d\n",pid);
        return 0;
}
~
```

이와 같이 system call 을 구현하였다.

ftrace 라는 system call 을 hooking 하여 다른 모듈의 함수로 대체할 것이기 때문에 간단하게 인자로 받은 pid 를 출력하게 끔만 구현하였다.

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

system call 을 컴파일하기 위한 Makefile 을 구성하고

os2019202050@ubuntu: ~/Downloads/linux-4.19.67

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

위와 같이 커널 컴파일시 컴파일하게끔 구현된 system call 함수가 있는 폴더를 등록해두고 커널을 컴파일 및 재부팅한다.

이제 336 system call ftrace 가 잘 동작하는지 확인한다.

test.c (~/hooking) - gedit

Open

abc.txt

```
#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);
        close(fd);
        syscall(336,0);
        return 0;

}
```

<test.c>

*abc.txt (~/hooking) -

Open

123456789

<abc.txt>

위의 파일을 과제의 검증용으로 사용할 예정이다. 위 c 파일을 컴파일한 후 실행하게 되면 syscall 에 의해 실행된 ftrace 가 c 파일을 실행하는 process 의 pid 를 출력하게 될 것이다.

```
os2019202050@ubuntu:~/hooking$ gcc -o test test.c
os2019202050@ubuntu:~/hooking$ ./test
os2019202050@ubuntu:~/hooking$ dmesg
```

```
[ 7061.408074] ftrace is called by pid:10027
[ 7061.408093] ftrace is called by pid:0
```

실행 후 dmesg 명령어를 통해 확인하면 10027 이라는 pid 값과 0 이라는 pid 값을 ftrace 가 잘 출력하는 것으로 보아 system call 이 올바르게 등록되었음을 확인할 수 있다.

이제는 open, close, read, write, lseek,ftrace system call 들을 모두 hooking 할 것이다.

먼저 ftracehooking.h 헤더파일이다.

```c
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/highmem.h>
#include <linux/kallsyms.h>
#include <linux/syscalls.h>
#include <asm/syscall_wrapper.h>
#include <asm/uaccess.h>
#include <linux/ktime.h>
```

모듈을 사용하기 위한 헤더파일, system call table 을 불러오기 위한 헤더파일, 시간출력을 위해 필요한 헤더파일 등등을 사용하기 위한 헤더파일들을 모아두었다.

ftracehooking.c 파일 중 함수 포인터 정의 부분과 변수 사용 목록이다.

```c
#include "ftracehooking.h"
#define __NR_ftrace 336

typedef asmlinkage int (*sys_call_ptr_t)(const struct pt_regs *);
sys_call_ptr_t *syscall_table;

sys_call_ptr_t real_ftrace;

pid_t proc_pid;
int open_count;
int close_count;
int read_count;
int write_count;
int lseek_count;
char file_name[100];
size_t read_bytes;
size_t written_bytes;
ktime_t nanoseconds;

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);
```

먼저 헤더파일을 불러오고 ftrace 의 system call number 는 336 이므로 define 하였다.

각각 pid 를 저장해 둘 변수, system call 들의 실행횟수를 저장해 둘 변수, open 을 통해 열람한 파일이름, 읽은 바이트 수, 쓴 바이트 수, trace 한 시간들이 저장될 변수들이 선언되어 있다.

그리고 각각의 변수들은 iotracehooking.c 에서 사용할 수 있게끔 EXPORT_SYMBOL 을 사용하였다.

추가적으로 typedef asmlinkage int (*sys_call_ptr_t)(const struct pt_regs *); 부분에 대한 설명이다.

이는 과제 조건중 const struct pt_regs *regs 를 사용하라는 것에 아이디어를 얻어 함수포인터를 정의해둔 문장이다. Cscope 를 통해 pt_regs 를 검색하니 아래와 같은 결과를 얻었고 이중 syscall.h 부분을 열어 확인해보니 다음과 같았다.

```
os2019202050@ubuntu: ~/Downloads/linux-4.19.67
Global definition: pt_regs

  File                 Line
0 processor-generic.h    9 struct pt_regs;
1 ptrace-generic.h      13 struct pt_regs {
2 kern_util.h           25 struct pt_regs;
3 bug.h                 15 struct pt_regs;
4 ptrace.h              49 struct pt_regs {
5 perf_event.h         524 int (*handle_irq)(struct pt_regs *);
6 extable.h             20 struct pt_regs;
7 kdebug.h               7 struct pt_regs;
8 kprobes.h             37 struct pt_regs;
9 nmi.h                 38 typedef int (*nmi_handler_t)(unsigned int , struct pt_regs *);
a perf_event.h         250 struct pt_regs;
b ptrace.h              12 struct pt_regs {
c ptrace.h              54 struct pt_regs {
d reboot.h               7 struct pt_regs;
e reboot.h              14 void (*crash_shutdown)(struct pt_regs *);
f reboot.h              28 typedef void (*nmi_shootdown_cb)(int , struct pt_regs*);
g signal.h             101 struct pt_regs;
h syscall.h             24 typedef asmlinkage long (*sys_call_ptr_t)(const struct pt_regs *);
i syscall_wrapper.h    204 struct pt_regs;
j ptrace.h              18 struct pt_regs {
k ptrace.h              44 struct pt_regs {
```

```
#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[];
```

X86_64 환경에서는 어셈블리 코드에서 직접 호출이 가능하고(asmlinkage) long 형태로 반환하며 pt_regs 라는 구조체를 인자로 받는 함수에 대한 포인터를 sys_call_ptr_t 라고 정의하였다. 또한 system call table 은 이러한 함수 포인터를 모아두는 테이블로 사용되었다.

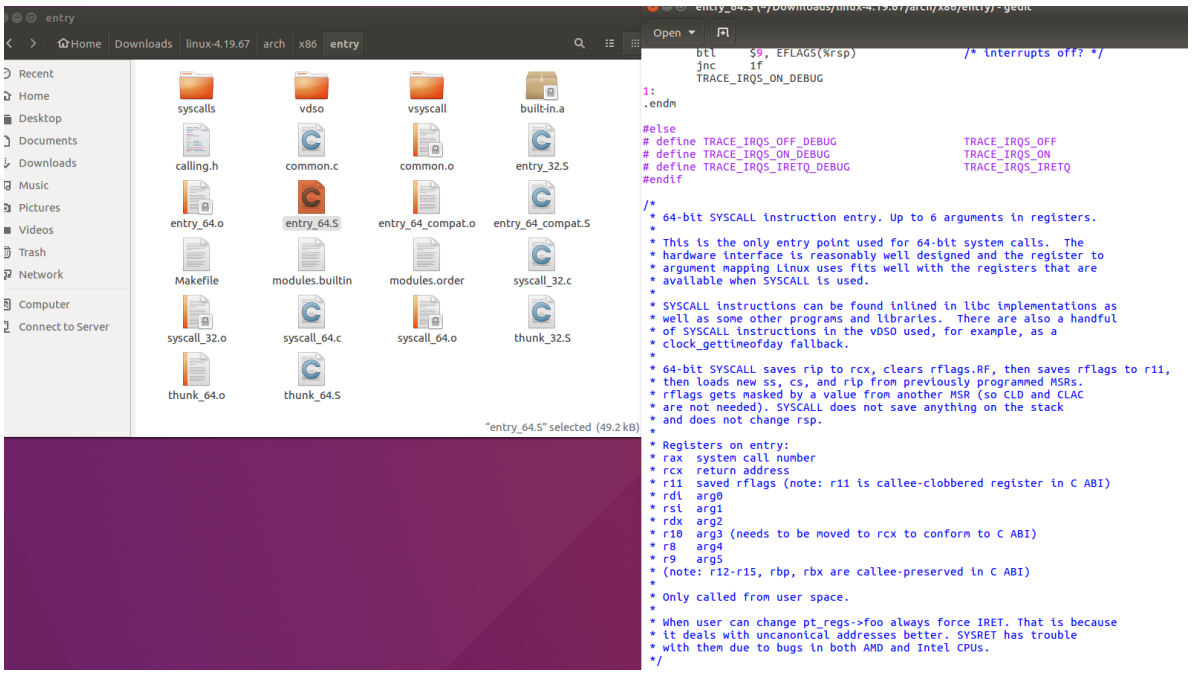
따라서 system call table 과 원래의 ftrace 를 저장해둘 변수로 sys_call_ptr_t 형태를 사용하였다.

과제 조건으로 ftrace 의 반환형은 int 이므로 sys_call_ptr_t 의 반환형도 int 로 맞춰주었다.

다음으로 pt_regs 에 대한 설명이다.

```
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
```

System call table 을 보면 위와 같이 system call 들이 x64 형태의 entry point 를 가진다고 한다.



위는 64-bit system call entry 에 대한 설명이다.

System call 함수들은 6 개의 인자까지 register 에 저장하며 rdi,rsi,rdx,r10,r8,r9 순으로 저장하며 이를 통해 system call 함수들의 원형을 확인하고 regs 와 -> 연산자를 통해 값을 얻을 수 있다.

여기까지의 설명으로 남은 코드를 보면

```
static asmlinkage int ftrace(const struct pt_regs *regs)
{
    if(regs->di != 0)
    {
        nanoseconds = ktime_get_ns();
        proc_pid = regs->di;
        open_count = 0;
        close_count = 0;
         read_count = 0;
        write_count = 0;
        lseek_count = 0;
        file_name[0] = '\0';
        read_bytes = 0;
        written_bytes = 0;
        printk("OS Assignment2 ftrace [%d] Start\n",proc_pid);
    }
    else
    {
        printk("[2019202050] /%s file[%s] stats [x] read - %d / written - %d\n",current -> comm,(char*)file_name,(int)read_bytes,(int)written_bytes);
        printk("[trace time: %llu ns] open[%d] close[%d] read[%d] write[%d] lseek[%d]\n",(ktime_get_ns()-nanoseconds),open_count,close_count,read_count,write_count,lseek_count);
        printk("OS Assignment2 ftrace [%d] End\n",proc_pid);
    }
    return 0;

}
```

syscall(336, pid)형태로 호출되는 함수이기 때문에 첫번째 인자에 접근해야 하므로 regs->di 를 통해 pid 값을 얻어낸다. 그 후 해당 값이 0 이 아니라면 pid 에 대한 추적 시작이므로 모든 변수들을 초기화 하고 추적 시작을 알린다.

0 이라면 여태껏 저장한 변수들의 값을 과제 출력 양식에 맞게 출력한다.

```
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;
}

static int __init hooking_init(void)
{
   syscall_table = (sys_call_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;
}

static 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");
```

make_rw 는 system call table 에 쓰기 권한을 부여하는 것이고 make_ro 는 회수하는 것이다. hooking_init 은 모듈이 적재되었을 때 실행되어 system call table 의 기존의 ftrace 를 잠시 저장해두고 대신 새롭게 구현한 ftrace 함수로 대체하는 것이다. hooking_exit 는 모듈이 언로드 되었을 때 기존의 ftrace 로 다시 돌려놓게끔 구현되어 있다.

다음으로 iotracehooking.c 이다.

```
#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 (*sys_call_ptr_t)(const struct pt_regs *);
sys_call_ptr_t *syscall_table;

sys_call_ptr_t real_open;
sys_call_ptr_t real_close;
sys_call_ptr_t real_read;
sys_call_ptr_t real_write;
sys_call_ptr_t real_lseek;

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

각각의 system call 함수들의 number 들을 define 해놓고 ftracehooking.c 에서 처럼 똑같이 함수포인터를 정의해 두었다.

그 후 system call 의 실행을 카운트하기 위해 외부변수를 사용하고 이때의 외부변수는 미리 ftracehooking.c 에 선언된 변수들이다. 따라서 iotracehooking.c 는 ftracehooking.c 에 의존성을 가짐을 알 수 있다.

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

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

Open 함수원형의 인자는 3 개이고 이때 filename 즉 첫번째 인자를 필요로 하기에 이를 가져와야 하고 이때 filename 은 사용자 영역, 모듈은 커널에 적재되어 있기에 strncpy_from_user 라는 데이터 교환함수를 사용하였고 regs->di 로 첫번째 인자에 접근하였다. 그후 open_count 값을 증가시켜 주었다.

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

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

Close 는 추가적으로 얻을 데이터가 없기에 close_count 만 증가시켜 주었다.

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

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

read 에서는 얼마나 읽었는지에 대한 정보가 필요하다. 필요한 인자는 세번째 인자 count 이고 따라서 regs->dx 를 통해 읽은 바이트 수에 접근하고 이를 외부변수에 더해주었다. 추가적으로 read_count 를 증가시켜 주었다.

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

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

Write 도 얼마나 썼는지에 대한 정보가 필요하므로 3 번째 인자에 해당하는 regs->dx 를 통해 얻은 바이트 수를 외부변수에 더해주었다. 추가적으로 write_count 도 증가시켰다.

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

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

Lseek 는 실행된 횟수만 카운트하면 된다. 따라서 lseek_count 만 증가시켰다.

위 5 개의 system call 을 대체할 함수들은 공통적으로 원래의 system call 들의 반환값을 반환한다. 그 이유는 다음과 같다.

```
ssize_t ksys_read(unsigned int fd, char __user *buf, size_t count)
{
        struct fd f = fdget_pos(fd);
        ssize_t ret = -EBADF;

        if (f.file) {
                loff_t pos = file_pos_read(f.file);
                ret = vfs_read(f.file, buf, count, &pos);
                if (ret >= 0)
                        file_pos_write(f.file, pos);
                fdput_pos(f);
        }
        return ret;
}

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

ssize_t ksys_write(unsigned int fd, const char __user *buf, size_t count)
{
        struct fd f = fdget_pos(fd);
        ssize_t ret = -EBADF;

        if (f.file) {
                loff_t pos = file_pos_read(f.file);
                ret = vfs_write(f.file, buf, count, &pos);
                if (ret >= 0)
                        file_pos_write(f.file, pos);
                fdput_pos(f);
        }

        return ret;
}

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

예를 들어 read 와 write 는 시스템 콜이 실행되면 인자들을 그대로 각각 ksys_read, ksys_write 에 전달하고 이 함수들의 반환값을 반환한다. 위의 5 개의 새로 정의한 함수들은 추가적으로 system call 들의 실행 횟수와 필요한 정보들을 추출하기 위해 table 을 조작하는 것일 뿐 원래 system call 로서의 기능을 다해야 한다. 따라서 기존의 system call 함수들도 실행하고 이를 반환값으로 하여 사용자 영역에서 동일하게 작동되어야 한다.

lotracehooking.c 또한 ftracehooking.c 와 마찬가지로 모듈의 로드 언로드때 system call table 의 권한을 부여하여 기존의 system call 을 저장해두고 위에서 설명한 함수들을 table 에 넣어두어 실행을 대체하였다.

이제 결과화면을 확인한다.

다음은 모듈 컴파일을 진행하고 모듈을 적재하는 과정이다.

```
Terminal
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
~
~
```

```
os2019202050@ubuntu:~/hooking$ sudo make
make -C /lib/modules/4.19.67-2019202050/build SUBDIRS=/home/os2019202050/hooking
 modules
make[1]: Entering directory '/home/os2019202050/Downloads/linux-4.19.67'
  CC [M]  /home/os2019202050/hooking/ftracehooking.o
  Building modules, stage 2.
  MODPOST 2 modules
  LD [M]  /home/os2019202050/hooking/ftracehooking.ko
make[1]: Leaving directory '/home/os2019202050/Downloads/linux-4.19.67'
os2019202050@ubuntu:~/hooking$ sudo insmod ftracehooking.ko
os2019202050@ubuntu:~/hooking$ sudo insmod iotracehooking.ko
os2019202050@ubuntu:~/hooking$ lsmod | grep hooking
iotracehooking         16384  0
ftracehooking          16384  1 iotracehooking
```

Iotracehooking 은 ftracehooking 모듈이 없으면 동작할 수 없기에 ftracehooking 먼저 로드해준다.

```
os2019202050@ubuntu:~/hooking$ gcc -o test test.c
os2019202050@ubuntu:~/hooking$ ./test
os2019202050@ubuntu:~/hooking$ dmesg
```

```
[ 1747.709945] OS Assignment2 ftrace [3716] Start
[ 1747.710033] [2019202050] /test file[abc.txt] stats [x] read - 20 / written -
20
[ 1747.710034] [trace time: 91205ns] open[1] close[1] read[4] write[4] lseek[9]
[ 1747.710035] OS Assignment2 ftrace [3716] End
```

위에서 언급한 test.c 를 실행시키니 test.c 를 실행시킨 pid, test 라는 실행파일명, 어떤 파일을 열람했는지, 읽고 쓴 바이트 수, 실행시간, 각 system call 을 실행한 횟수 등이 올바르게 나오는 것을 확인했다.

또한 기존의 system call 의 역할인 문자열 복사도 수행하였다.

```
                                                    abc.txt
123456789
123456789
```

```
os2019202050@ubuntu:~/hooking$ lsmod |grep hooking
iotracehooking         16384  0
ftracehooking          16384  1 iotracehooking
os2019202050@ubuntu:~/hooking$ sudo rmmod iotracehooking
[sudo] password for os2019202050:
os2019202050@ubuntu:~/hooking$ sudo rmmod ftracehooking
os2019202050@ubuntu:~/hooking$ lsmod |grep hooking
os2019202050@ubuntu:~/hooking$ ./test
os2019202050@ubuntu:~/hooking$ dmesg
```

모듈을 제거하니 아래와 같이 기존의 system call 이 다시 수행된다.

```
[ 2114.827177] ftrace is called by pid:3770
[ 2114.827244] ftrace is called by pid:0
```

3. 고찰

늘 라이브러리를 통해 코딩하고 과제작업을 진행했는데 system call 이라는 것을 직접 커널 레벨에 접근하여 만들어보고 사용해보니 이론적으로 배웠던 것들이 더 확실히 이해가 되는 느낌이었다. 하지만 로우레벨에서 작성되는 코드이니 훨씬 민감하고 잘못 프로그래밍되면 동작이 쉽게 멈추는 경우가 많았다. 실습을 기반으로 system call 을 만들었는데 system call table 의 중간 번호가 비어있어 코드의 중간에 함수를 작성하였더니 부팅이 되지 않았는데 이는 64 비트 엔트리를 가진 system call 과 32 비트 엔트리를 가진 system call 를 나누는 경계였고 이 부분은 64 비트 system call 들의 cache 구역으로 사용된다는 주석을 나중에 읽고 나서야 수정하게 되었다. 이처럼 코드들이 매우 면밀하게 구성되어 있어 정말 사소한 것에서도 오류 발생하였다.

```c
static void __exit hooking_exit(void)
{
   syscall_table[__NR_open] = real_open;
   syscall_table[__NR_close] = real_close;
   syscall_table[__NR_read] = real_read;
   syscall_table[__NR_write] = real_write;
   syscall_table[__NR_lseek] = real_lseek;
   make_ro(syscall_table);
}
```

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

또한 위 처럼 모듈이 제거가 되지 않는 경우가 있었는데 이는 실습 코드에 주어진 방식으로는 모듈을 언로드할 때 테이블을 수정할 수 있는 권한을 도로 회수하는 코드가 있었는데 iotracehooking 이 이를 회수한 후 언로드하여 ftracehooking 이 언로드할 때 테이블을 수정하지 못하여 원래의 system call 로 되돌릴 수 없어 생기는 문제였다. 세마포어에서 락이 걸리는 것처럼 이러한 수정권한을 잘못 설정하게 되면 생길 수 있는 문제들과 모듈 간의 의존성 문제가 쉽게 생길 수 있다는 점이 커널레벨에서의 프로그래밍에 신중을 기할 수 밖에 없게 만들었던 것 같다.

4. Reference

운영체제실습 강의자료 참조

https://www.kernel.org/doc/html/latest/core-api/timekeeping.html - nanosecond 출력