

Assignment 4

수업 명:	운영체제
담당 교수님:	최상호 교수
학번:	2018202074
이름:	김상우
강의시간:	금 1,2

Introduction

해당 과제는 이전처럼 wrapping을 이용해 ftrace를 대체하며 해당 과정에서 task_struct를 사용하며 이를 통해 PID를 바탕으로 프로세스의 정보들을 뽑아낸다. 이를 통해 가상 메모리에 파일의 정보가 저장됨과 출력방법을 알아낸다. 또한 Dynamic Recompilation을 사용하여 C언어 환경에서 Asm파일을 어떻게 사용하는 지 알며, Shared memory를 이용해 특정 데이터를 파일간 공유하는 방법을 알아보고 어셈블리를 최적화하는 방법도 고안해 이를 제작한다. 해당 과정에서 권한 제어와 mmap의 사용을 익히는 것도 목표로 하고 있다.

Conclusion & Analysis

Assignment #4-1

코드 작성에 앞서 아래와 같이 assignment 2에서 작성한 ftrace를 확인한다.

```
os2018202074@ubuntu:~/Downloads/linux-4.19.67$ vi arch/x86/entry/syscalls/syscall_64.tbl
```

```
331      common  pkey_free      __x64_sys_pkey_free
332      common  statx          __x64_sys_statx
333      common  io_pgetevents __x64_sys_io_pgetevents
334      common  rseq          __x64_sys_rseq
336      common  ftrace        __x64_sys_ftrace
#
```

위를 통해 syscall 함수에 인자로 336을 넣어 해당 함수를 불러올 수 있음을 확인했다.

```
os2018202074@ubuntu:~/Downloads/linux-4.19.67$ vi include/linux/syscalls.h
```

```
asmlinkage long sys_hello(void);
asmlinkage long sys_add(int, int);
asmlinkage int ftrace(pid_t pid);
#endif
```

위같이 기존의 함수는 pid_t pid를 인자로 받으며 ftrace라는 이름을 가진 것을 확인할 수 있었다.

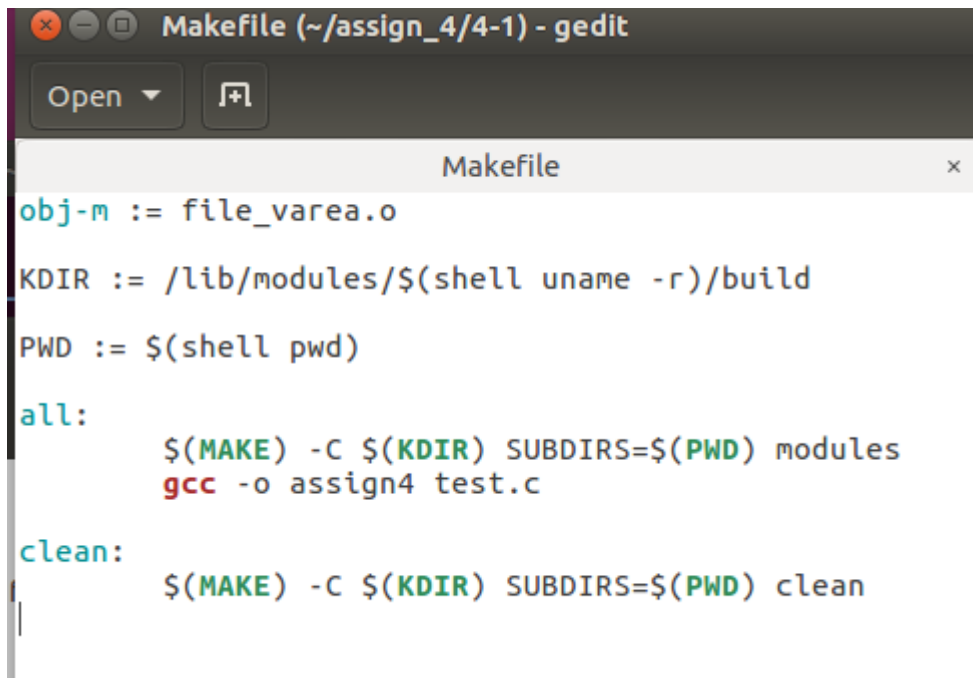
```
os2018202074@ubuntu:~/Downloads/linux-4.19.67$ cd ftrace
os2018202074@ubuntu:~/Downloads/linux-4.19.67/ftrace$ vi ftracehooking.c
```

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

SYSCALL_DEFINE1(ftrace, pid_t, pid)
{
    printk("No hijack %ld\n", pid);
    return 0;
}
~
```

또한 해당 system call은 printk를 통해 pid와 함께 위 같은 문자열을 출력하고 0을 return하는 것을 확인했다.

위와 같이 wrapping을 진행할 함수를 확인하였다.



```
obj-m := file_varea.o

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

PWD := $(shell pwd)

all:
    $(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules
    gcc -o assign4 test.c

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

Make file의 코드는 위와 같다. 조건에 맞추어 Assign4라는 이름의 테스트용 실행파일을 주어진 test.c 코드를 기반으로 생성한다. 또한 file_varea를 이용할 수 있도록 하는 것을 확인할 수 있다.

-file_varea.c

```
#include <linux/module.h>
#include <linux/highmem.h>
#include <linux/kallsyms.h>
#include <linux/syscalls.h>

#include <asm/syscall_wrapper.h>
#include <linux/kernel.h>
#include <linux/unistd.h>
#include <linux/sched.h>
#include <linux/init_task.h>

#include <linux/sched/mm.h>
#include <linux/string.h>

#define __NR_ftrace 336

void ** syscall_table; //save system call table
void *real_ftrace; //save original ftrace function
```

코드를 작성하기 위한 헤더들과 함께 syscall table을 담을 변수, 그리고 원래의 ftrace를 저장할 변수를 선언한다.

```
void make_rw(void *addr) //function to read and write
{
    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) //function to cancel (read and write)
{
    unsigned int level;
    pte_t * pte = lookup_address((u64)addr, &level);
    pte->pte = pte->pte &~ _PAGE_RW;
}
```

addr이 속해 있는 페이지에 읽기 및 쓰기 권한을 부여하는 함수 make_rw를 위와 같이 선언하고 이를 통해 system call table에 대해 쓰기 권한을 취득하여 해당 model이 hijack할 수 있게 한다. Make_ro를 통해서는 이를 원래대로 돌린다.

```
__SYSCALL_DEFINE4(1, file_varea, pid_t, pid){//Hooking function(name: file_varea)
    int i;
    struct task_struct *task1;    //take task_struct based pid
    task1 = pid_task(find_vpid(pid), PIDTYPE_PID);

    struct mm_struct *mm;    //mm_struct to data/code/heap address of process
    mm = get_task_mm(task1);

    struct vm_area_struct *vm;    //process name and memory address
    vm = mm->vmmap;

    unsigned long mem_start;    //variable to save address of memory, data, code, heap
    unsigned long mem_end;

    unsigned long data_start;
    unsigned long data_end;
    data_start = mm->start_data;
    data_end = mm->end_data;

    unsigned long code_start;
    unsigned long code_end;
    code_start = mm->start_code;
    code_end = mm->end_code;

    unsigned long heap_start;
    unsigned long heap_end;
    heap_start = mm->start_brk;
    heap_end = mm->brk;
}
```

해당 함수는 file_varea라는 이름을 가져야 하고 pid를 인자로 받아야 하기에 위와 같이 선언한다. 이후 pid를 기반으로 task_struct에 맞춰 정보를 받아오고, 해당 정보를 바탕으로 mm_struct와 vm_area_struct를 얻어낸다. 또한 이들을 기반으로 정보가 위치하는 가상 메모리 주소를 저장할 변수를 선언하고, 프로세스의 데이터 주소, 코드 주소, 힙 주소를 저장할 변수 선언 및 값 입력을 진행한다.

```
    int map_count;
    map_count = mm->map_count;

    char * path;
    printk("##### Loaded files of a process '%s(%d)' in VM #####\n", task1->comm, pid);
    //print process name & pid
    for(i = 0; i < map_count && vm != NULL; i++){    //check all them
        path = vm->vm_file->f_path.dentry->d_iname;    //take path of file
        mem_start = vm->vm_start;    //memory address check
        mem_end = vm->vm_end;

        if(vm->vm_file != NULL){    //print information
            printk("mem(%lx~%lx) code(%lx~%lx) data(%lx~%lx) heap(%lx~%lx) %s\n",
                mem_start, mem_end, code_start, code_end, data_start, data_end, heap_start,
                path);
        }
        vm = vm->vm_next;    //go to next
    }
    printk("#####\n");
    return 0;
}
```

이후 확인할 대상의 수를 파악하기 위해 map_count를 정의하고 이후 원본 파일 위치를 파악하기 위해 path를 선언한다. 양식에 맞게 정보들을 출력하며 해당 과정에서 map_count를 이용, 범위 내에서 vm_area_struct->next를 이용하여 각 정보에 대해 출력하는 모습이다.

```

static int __init hooking_init(void)    //take and save original system call
{
    syscall_table = (void **) kallsyms_lookup_name("sys_call_table");
    //save system call table
    make_rw(syscall_table);
    real_ftrace = syscall_table[__NR_ftrace]; //save original function
    syscall_table[__NR_ftrace] = __x64_sysfile_varea;
    return 0;
}
static void __exit hooking_exit(void)    //return to original function
{
    syscall_table[__NR_ftrace] = real_ftrace; //return original function
    make_ro(syscall_table);
}
module_init(hooking_init);
module_exit(hooking_exit);
MODULE_LICENSE("GPL");

```

이후 해당 함수들에 대한 대체가 이뤄지기 전 처리를 위해 __init hooking_init를 사용, syscall table 내에 kallsyms_lookup_name을 통해 기존에 사용하던 sys_call_table, 즉 system call에 대한 표를 받아온다. 이후 real_ftrace 내에 원래 위치에 있던 함수를 syscall_table[system call number]를 통해 저장하고, syscall_table[__NR_ftrace]의 위치에 대체할 함수 file_varea를 넣는 것을 볼 수 있다.

이는 __exit hooking_exit를 통해 원래대로 돌릴 필요가 있다. 이를 위해 syscall_table[__NR_ftrace] 위치에 이전에 real_ftrace에 저장했던 원래 함수를 다시 넣어준다.

이후 module_init, module_exit를 통해 위 코드 기반의 모듈의 각 생성/종료때 각 함수를 실행한다.

Module_license("GPL")을 통해 모듈의 라이선스는 GPL이 된다.

위는 numgen.c를 위해 작성한 코드이다. 위를 보면, temp.txt파일을 fopen을 통해 생성하고 w, 즉 쓰기 모드로 이를 엽니다. 이후 MAX_PROCESS의 두배만큼 fprintf를 이용해 숫자를 순서대로 적습니다. 이후 f_close를 이용해, 파일을 닫습니다.

```

os2018202074@ubuntu:~/assign_4/4-1$ make
make -C /lib/modules/4.19.67-os2018202074/build SUBDIRS=/home/os2018202074/assign_4/4-1 modules
make[1]: Entering directory '/home/os2018202074/Downloads/linux-4.19.67'
  CC [M]  /home/os2018202074/assign_4/4-1/file_varea.o
/home/os2018202074/assign_4/4-1/file_varea.c: In function '__do_sysfile_varea':
/home/os2018202074/assign_4/4-1/file_varea.c:40:2: warning: ISO C90 forbids mixed declarations and code [-Wdeclaration-after-statement]
    struct mm_struct *mm;
    ^
/home/os2018202074/assign_4/4-1/file_varea.c:43:2: warning: ISO C90 forbids mixed declarations and code [-Wdeclaration-after-statement]
    struct vm_area_struct *vm;
    ^
/home/os2018202074/assign_4/4-1/file_varea.c:46:2: warning: ISO C90 forbids mixed declarations and code [-Wdeclaration-after-statement]
    unsigned long mem_start;
    ^
/home/os2018202074/assign_4/4-1/file_varea.c:54:2: warning: ISO C90 forbids mixed declarations and code [-Wdeclaration-after-statement]
    unsigned long code_start;
    ^

```

위같이 작성된 파일을 바탕으로 make를 진행하였다.

```
gcc -o assign4 test.c
os2018202074@ubuntu:~/assign_4/4-1$ ls
assign4      file_varea.ko      file_varea.mod.o   Makefile          Module.symvers
file_varea.c file_varea.mod.c   file_varea.o       modules.order     test.c
os2018202074@ubuntu:~/assign_4/4-1$
```

위와 같은 파일들이 생성됨을 확인할 수 있었다.

```
os2018202074@ubuntu:~/assign_4/4-1$ ./assign4
os2018202074@ubuntu:~/assign_4/4-1$ dmesg
```

```
[ 7256.571804] No hijack 4530
os2018202074@ubuntu:~/assign_4/4-1$
```

원래 function이 나오는 것을 확인했다. 이로서 환경이 적절하게 setting되었음을 알 수 있었다.

```
os2018202074@ubuntu: ~/assign_4/4-1
os2018202074@ubuntu:~/assign_4/4-1$ sudo insmod file_varea.ko
[sudo] password for os2018202074:
os2018202074@ubuntu:~/assign_4/4-1$ ./assign4
os2018202074@ubuntu:~/assign_4/4-1$ sudo rmmod file_varea
os2018202074@ubuntu:~/assign_4/4-1$ dmesg
```

이후 위와 같은 과정을 통해 작성이 잘 되었는지 확인한다.

```
7320.267365 ##### Loaded files of a process 'assign4(4543)' in VM #####
7320.267366 mem(400000~401000) code(400000~40074c) data(600e10~601040) heap(8c9000~8c9000) assign4
7320.267367 mem(600000~601000) code(400000~40074c) data(600e10~601040) heap(8c9000~8c9000) assign4
7320.267367 mem(601000~602000) code(400000~40074c) data(600e10~601040) heap(8c9000~8c9000) assign4
7320.267368 mem(7f9f30086000~7f9f30246000) code(400000~40074c) data(600e10~601040) heap(8c9000~8c9000) libc-2.23.so
7320.267368 mem(7f9f30246000~7f9f30446000) code(400000~40074c) data(600e10~601040) heap(8c9000~8c9000) libc-2.23.so
7320.267369 mem(7f9f30446000~7f9f3044a000) code(400000~40074c) data(600e10~601040) heap(8c9000~8c9000) libc-2.23.so
7320.267369 mem(7f9f3044a000~7f9f3044c000) code(400000~40074c) data(600e10~601040) heap(8c9000~8c9000) libc-2.23.so
7320.267370 mem(7f9f30450000~7f9f30476000) code(400000~40074c) data(600e10~601040) heap(8c9000~8c9000) ld-2.23.so
7320.267370 mem(7f9f30675000~7f9f30676000) code(400000~40074c) data(600e10~601040) heap(8c9000~8c9000) ld-2.23.so
7320.267371 mem(7f9f30676000~7f9f30677000) code(400000~40074c) data(600e10~601040) heap(8c9000~8c9000) ld-2.23.so
7320.267371 #####
os2018202074@ubuntu:~/assign_4/4-1$
```

원하는 결과가 출력되는 것을 확인했다.

Assignment #4-2

```
os2018202074@ubuntu: ~/assign_4/4-2-2
os2018202074@ubuntu:~/assign_4/4-2-2$ ls
D_recompile.c D_recompile_test.c Makefile
os2018202074@ubuntu:~/assign_4/4-2-2$ gcc -c D_recompile_test.c
D_recompile_test.c: In function 'Operation':
D_recompile_test.c:59:2: warning: missing terminating " character
    "imul %%eax, 2;
    ^
D_recompile_test.c:59:2: error: missing terminating " character
D_recompile_test.c:76:9: warning: missing terminating " character
    "div %%dl;
    ^
D_recompile_test.c:60:2: error: expected ':' or ')' before string constant
    "imul %%eax, 2;"
    ^
D_recompile_test.c:76:9: error: missing terminating " character
    "div %%dl;
    ^
```

코드를 실행하기 앞서 objdump를 통해 Dump를 출력해보고자 했다. 그러나 내부에 error가 존재, 이를 수정하기로 한다.

```

"sub %%eax, 1;"
"imul %%eax, 2;"
"imul %%eax, 2;"
"imul %%eax, 2;"
"add %%eax, 1;"
"add %%eax, 1;"
"add %%eax, 3;"
"add %%eax, 1;"
"add %%eax, 1;"
"add %%eax, 1;"
"add %%eax, 3;"
"add %%eax, 1;"
"add %%eax, 1;"
"add %%eax, 2;"
"add %%eax, 1;"
"add %%eax, 1;"
"add %%eax, 1;"
"add %%eax, 1;"
"add %%eax, 1;"
"div %%dl;"
"div %%dl;"
"sub %%eax, 1;"

```

```

"sub %%eax, 1;"
"imul %%eax, 2;"
"imul %%eax, 2;"
"imul %%eax, 2;"
"add %%eax, 1;"
"add %%eax, 1;"
"add %%eax, 3;"
"add %%eax, 1;"
"add %%eax, 1;"
"add %%eax, 1;"
"add %%eax, 3;"
"add %%eax, 1;"
"add %%eax, 1;"
"add %%eax, 2;"
"add %%eax, 1;"
"add %%eax, 1;"
"add %%eax, 1;"
"add %%eax, 1;"
"add %%eax, 1;"
"div %%dl;"
"div %%dl;"
"sub %%eax, 1;"

```

D_recompile_test.c 파일 내 큰따옴표가 안적힌 부분을 발견할 수 있었다. 이를 왼쪽에서 오른쪽 캡처로 변경하였다.

```

os2018202074@ubuntu: ~/assign_4/4-2
os2018202074@ubuntu:~/assign_4/4-2$ ls
D_recompile.c D_recompile_test.c Makefile
os2018202074@ubuntu:~/assign_4/4-2$ gcc -c D_recompile_test.c
os2018202074@ubuntu:~/assign_4/4-2$ objdump -d D_recompile_test.o > test
os2018202074@ubuntu:~/assign_4/4-2$ ls
D_recompile.c D_recompile_test.c D_recompile_test.o Makefile test
os2018202074@ubuntu:~/assign_4/4-2$

```

이후 에러 없이 위 코드가 실행되었다. Objdump 명령어를 이용해 D_recompile_test.c를 통해 생성되는 dump에 대한 정보를 가진 test파일을 생성하는 모습이다.

```

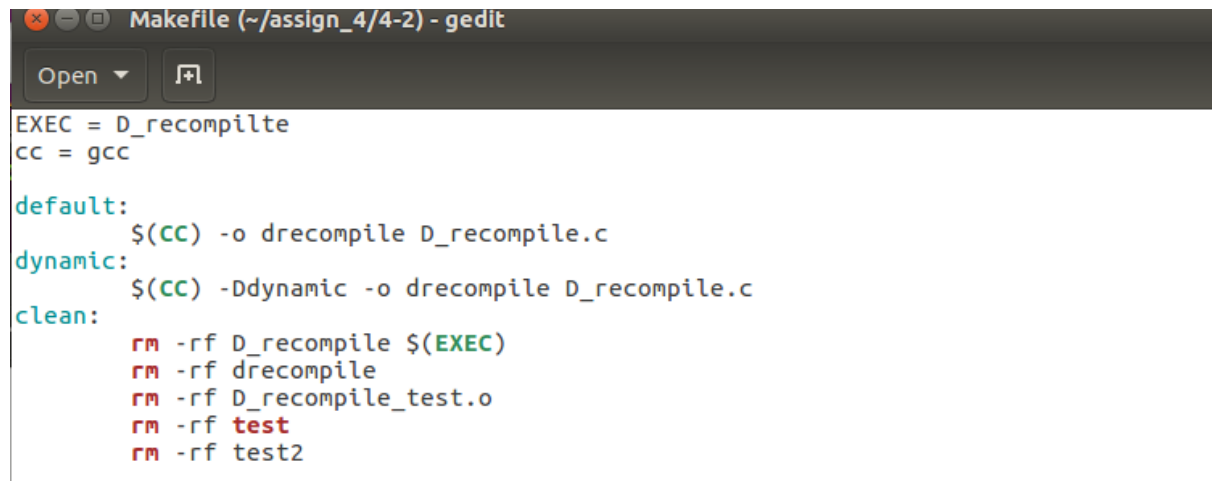
test (~/assign_4/4-2) - gedit
Open [icon]
Makefile x test x *file_varea.c x
D_recompile_test.o:      file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <Operation>:
0:  55                push    %rbp
1:  48 89 e5          mov     %rsp,%rbp
4:  89 7d fc          mov     %edi,-0x4(%rbp)
7:  8b 55 fc          mov     -0x4(%rbp),%edx
a:  89 d0             mov     %edx,%eax
c:  b2 02            mov     $0x2,%dl
e:  83 c0 01          add     $0x1,%eax
11: 83 c0 01          add     $0x1,%eax
14: 83 c0 01          add     $0x1,%eax
17: 83 c0 01          add     $0x1,%eax
1a: 83 c0 02          add     $0x2,%eax
1d: 83 c0 03          add     $0x3,%eax
20: 83 c0 01          add     $0x1,%eax
23: 83 c0 02          add     $0x2,%eax
26: 83 c0 01          add     $0x1,%eax
29: 83 c0 01          add     $0x1,%eax
2c: 6b c0 02          imul    $0x2,%eax,%eax
2f: 6b c0 02          imul    $0x2,%eax,%eax
32: 6b c0 02          imul    $0x2,%eax,%eax
35: 83 c0 01          add     $0x1,%eax
38: 83 c0 01          add     $0x1,%eax
3b: 83 c0 03          add     $0x3,%eax
3e: 83 c0 01          add     $0x1,%eax
41: 83 c0 01          add     $0x1,%eax
44: 83 c0 01          add     $0x1,%eax
47: 83 c0 03          add     $0x3,%eax
4a: 83 c0 01          add     $0x1,%eax
4d: 83 c0 01          add     $0x1,%eax
50: 83 c0 02          add     $0x2,%eax
53: 83 c0 01          add     $0x1,%eax
56: 83 c0 01          add     $0x1,%eax
59: 83 c0 01          add     $0x1,%eax

```


해당 파일내에 Dump가 잘 저장됨을 확인했다. 내부에는 D_recompile_test.c내 Operation으로 정의되어 있는 asm 코드와 동일한 내용이 기술되어 있었으며, 해당 asm에 대해 16진수로 어떤식으로 입력되는 지 적혀 있었다. 이를 확인하면 위 Operation에서 각 함수가 hexadecimal 형태로 사용되는 지 알 수 있다. 이를 바탕으로 add, sub, imul, div를 감지하고 다른 operation에 대해 위 hexadecimal 정보를 배열의 형태로 저장해 return하여 함수처럼 사용할 수 있다는 것을 도출해 내었다. 동시에 dl register 내에 고정값으로 무엇이 들어가는 지 알 수 있었다.(mov \$0x2, \$dl을 통해 dl register에 고정값으로 2가 들어가고 있음을 알 수 있다. 코드에선 이후 dl register의 값을 변화시키지 않음도 확인할 수 있었다.)



```

EXEC = D_recompilte
cc = gcc

default:
$(CC) -o drecompile D_recompile.c
dynamic:
$(CC) -Ddynamic -o drecompile D_recompile.c
clean:
rm -rf D_recompile $(EXEC)
rm -rf drecompile
rm -rf D_recompile_test.o
rm -rf test
rm -rf test2

```

이후 makefile을 생성하였다. 위 makefile을 통해 drecompile이라는 이름의 출력 파일로서 D_recompile.c를 사용할 수 있다. 이때 make / make dynamic을 입력하는 여부에 따라 결과가 달라지게 하기 위해 위와 같이 코드를 작성하였다.

-D_recompile_test.c

```

#include <stdio.h>
#include <stdint.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/user.h>

int Operation(int a)
{
    __asm__(
        ".intel_syntax;"
        "mov %%eax, %1;"
        "mov %%dl, 2;"
        "add %%eax, 1;"
        "add %%eax, 1;"
        "add %%eax, 1;"
        "add %%eax, 1;"
        "add %%eax, 2;"
        "add %%eax, 3;"
        "add %%eax, 1;"
        "add %%eax, 2;"
        "add %%eax, 1;"
        "add %%eax, 1;"
    )
}

```

해당 파일은 Operation이라는 이름하에 asm형태의 Operation들이 저장되어 있다.


```

int main(void)
{
    uint8_t* func = (uint8_t*)Operation;
    int i = 0;
    int segment_id;
    uint8_t* shared_memory;
    segment_id = shmget(1234, PAGE_SIZE, IPC_CREAT | S_IRUSR | S_IWUSR);
    shared_memory = (uint8_t*)shmat(segment_id, NULL, 0);
    do
    {
        shared_memory[i++] = *func;
    } while (*func++ != 0xC3);

    shmdt(shared_memory);
    printf("Data was filled to shared memory.\n");
    return 0;
}

```

Windows 정

이러한 정보는 위의 코드에서 확인이 가능하듯이 shmget을 통해 위와 같은 조건하에 shared memory로서 공유되고 있다. 위를 보면 이러한 shared_memory영역에 Operation 내에 있던 asm 함수들을 저장하는 모습이다.

-D_recompile.c

```

#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/user.h>
#include <sys/mman.h>

#include <string.h>          // strlen()
#include <fcntl.h>           // O_WRONLY
#include <unistd.h>

#include <sys/time.h>        // calculate time
#include <time.h>
uint8_t* Operation;
uint8_t* compiled_code;

void sharedmem_init(); //shared memory initialize
void sharedmem_exit(); // shared memory free
void drecompile_init(); // to compiled_code initialize
void drecompile_exit(); // to compiled_code free
void* drecompile(uint8_t *func); //func file make(asm array)

uint8_t * shared_memory; //take place to save shared memory

```

위에서 언급된 shared memory를 통해 공유된 정보를 받아 실행하기 위한 헤더(시간 측정을 위한 헤더도 선언하였다.)와 함수, 변수들을 선언한다. 최종적으로 compiled_code에 shared memory값을 받고 Operation에 최종 함수들을 옮겨 실행할 수 있게 할 것이다.

```

int len=0;           //length of asm array

int pagesize;       //page size
int k=0;             //save Operation place
int f=0;             //save compiled_code place

int main(void)
{
    int (*func)(int a);
    int i;

    sharedmem_init();
    drecompile_init(shared_memory);

    func = (int (*)(int a))drecompile(Operation);

    struct timespec startTime, endTime;    //variable to save time
    long diffTime=0;                       //total run time of func(1)
    long subdiffTime=0;                   //each run time of func(1)

    int result=0;
    int subresult=0;

    for(int l=0;l<50;l++){
        system("sync");
        system("echo 3 | sudo tee /proc/sys/vm/drop_caches");
        clock_gettime(CLOCK_MONOTONIC, &startTime); //start of time
        result=func(1);
        clock_gettime(CLOCK_MONOTONIC, &endTime); //end of time
        subdiffTime=1000000000 * (endTime.tv_sec - startTime.tv_sec)+(endTime.tv_nsec-
startTime.tv_nsec);
        printf("%d(th) run time: %lf\n", l,(double)subdiffTime); //print time(nano-sec)
        diffTime += subdiffTime;
    }
    printf("\ntotal run time: %lf\n", (double)diffTime); //print time(nano-sec)
    printf("average run time: %lf\n", (double)diffTime/50); //print time(nano-sec)
    printf("result: %d\n", result); //check asm right

    //printf("total execution time: %lf sec\n", (double)subdiffTime/1000000000); //print time(nano-sec)

    drecompile_exit();
    sharedmem_exit();
    return 0;
}

```

추가적으로 이후 저장 위치를 나타내기 위해 사용될 변수와 총 길이, page size를 저장할 변수를 선언한다. 주어진 Main의 경우 함수 포인터를 선언하고 shared memory와 drecompile, 즉 데이터를 받아올 준비를 한 후 drecompile을 통해 asm 정보를 모두 받아오고 이를 func(1)을 통해 1set 씩 실행한다. 이때 총 50만번 이를 진행하게 되며 실행시 마다 캐시 및 버퍼를 지워 실험에 영향을 주는 요소를 제거한다. 또한 각 실행에 대해 func(1)의 시간을 측정해 출력하고 이들의 평균을 내는 모습을 볼 수 있다. 추가적으로 결과가 맞는 지 확인하기 위해 result값도 출력해주었다.

이후 objdump를 이용해 test파일 내에 해당 정보를 저장할 수 있도록 하였다.

```

void sharedmem_init()
{
    int id_seg = shmget(1234, PAGE_SIZE, 0); //take ID to shared memory connect
    shared_memory = (uint8_t*)shmat(id_seg, NULL, 0); //connected array
}

void sharedmem_exit()
{
    shmdt(shared_memory); //disconnect shared memory
}

```

Sharememory에 대해서는 D_recompile_test.c에서 Operation 정보를 넘겨준 곳에 접근하기 위해 동일한 위치를 접근하기 위한 ID를 shmget을 통해 받아오고 이를 바탕으로 shared_memory 변수를 통해 해당 공간에 접근할 수 있도록 shmat를 사용하였다. Share memory에 대한 접근을 멈출 시 shared된 곳에 대한 접근을 다시 막기위해 shmdt를 이용한다.

```

void drecompile_init(uint8_t *func)
{
    do{
    }while(shared_memory[len++] != 0xC3); //take length of shared memory
    //fd = open("./hello.txt", O_RDWR|O_CREAT, 0777);
    //char fun_val[5];
    pagesize = getpagesize(); //take page size
    compiled_code = (uint8_t *)mmap(0, pagesize, PROT_READ | PROT_WRITE, MAP_ANONYMOUS |
MAP_SHARED, -1, 0);
    //compiled_code take memory by using mmap
    k=0;
    do{
        compiled_code[k] = shared_memory[k];
        //compiled code take shared memory's data
    }while(k++<len);

    msync(compiled_code, pagesize, MS_ASYNC);
    //map synchronize
}

void drecompile_exit()
{
    munmap(compiled_code, pagesize); //free mmaped variable
    munmap(Operation, len);
    //free mapped Operation
}

```

Compiled을 위한 setting의 경우 shared memory를 통해 받아오는 코드의 크기를 변수로 받고 page size는 getpagesize를 이용해 구해낸다. 이후 compiled_code에 메모리를 할당(이때 fd를 쓰는 것이 아닌 MAP_ANONYMOUS를 이용해 받아왔다. 때문에 fd 자리에 -1이 사용되었다.)하고 Read Write 권한이 부여된 만큼 compiled_code내에 shared_memory로부터 받아온 asm 명령어들을 복사한다. 이후 mmap에 대해 synchronize를 진행하는 모습이다.

해제시에는 mmap를 진행할 Compiled_code와 Operation에 대해 mmap을 해제할 것이다.

```

void drecompile(uint8_t *func)
{
#ifdef dynamic //when make dynamic
    pagesize = getpagesize(); //get page size
    Operation = (uint8_t *)mmap(0, pagesize, PROT_READ | PROT_WRITE, MAP_ANONYMOUS | MAP_SHARED, -1, 0);
    //map Operation to take asm data(read and write)
    k=0; //Operation place
    f=0; //Compiled_code place
    int can_1=0; //save current function
    int can_2=0;
    int can_3=0; //save current value
    do{
        if(can_1 == compiled_code[f] && can_2 == compiled_code[f+1]){ //if instruction same
            if(can_1 == 0xf6 && can_2 == 0xf2){ //div case
                can_3 *= 2; //put square of dl value(2) to reduce number of div
                f += 2; //read control
            }
            else if(can_1 == 0x83 && can_2 == 0xc0){ //add case
                can_3 += compiled_code[f+2]; //add value to reduce number of add
                f += 3;
            }
            else if(can_1 == 0x83 && can_2 == 0xe8){ //sub
                can_3 += compiled_code[f+2]; //add value to reduce number of sub
                f += 3;
            }
            else if(can_1 == 0x6b && can_2 == 0xc0){ //imul
                can_3 *= compiled_code[f+2]; //multiply value to reduce number of imul
                f += 3;
            }
        }
        else if(compiled_code[f] != 0x92 && compiled_code[f] != 0x66 && compiled_code[f] != 0xf6){

```

Drecompile 함수에선 Operation배열 내에 최종적으로 값이 들어가 실행가능한 상태로 만들어질 것이다. 만약 make dynamic을 입력 받을 경우 ifdef dynamic에 의해 다음 endif까지의 코드만이 해당 함수동안 유효하게 될 것이다. Make dynamic에 의해 진행되는 경우는 Asm 코드를 저장하고 있는 Compiled_code와 위 정보를 적게 될 Operation에서의 위치를 설정하기 위해 f와 k를 0으로 초기화 한다. 또한 최적화를 위해 중복여부를 확인하기 위해 명령어 종류와 사용되는 value를 저장하기 위한 can_1, can_2, can_3 변수를 선언한다.

Operation의 경우 compiled_code의 값을 받을 것이기에 mmap를 통해 메모리를 할당받는다. 이때 값이 수정되므로 Write권한을 갖게 설정되었다.(Read- Write) 또한 위와 마찬가지로 MAP_ANONYMOUS를 이용해 fd대신 -1을 인자로 받고 있다.

만약 저장하고 있는 명령어의 종류와 다음 명령어가 같다면 아래와 같은 처리를 진행한다.

만약 dl register에 대한 div가 진행될 경우 value를 저장하는 can_3에 dl의 고정값인 2를 곱해 저장하고 읽는 위치를 2칸 옮긴다는 의미인 f에 2를 더하는 행위를 진행한다.

만약 add나 sub가 들어올 경우 더하거나 뺄 값에 이를 더해주어 한번에 할 수 있도록 최적화를 진행한다. Imul이 입력되는 경우도 value에 곱셈을 진행하는 것을 제외하면 마찬가지이다. 이때 add/ sub/ imul /div 판단을 위해 사용되는 hexa숫자의 경우 odjdump를 통해 생성된 text파일을 통해 파악할 수 있었다.

```
else if(compiled_code[f] != 0xb3 && compiled_code[f] != 0xb0 && compiled_code[f] != 0xb7){
    //no add/sub/imul/div operation
    if(can_1 != 0x00 && can_2 != 0x00){ //if current operation exist
        if(can_1 == 0xf6 && can_2 == 0xf2){ //div case
            if(can_3 >= 4){ //when overlap div exist over two time
                Operation[k] = 0xb2; //put current value in dl register
                Operation[k+1] = can_3;
                k+=2; //write control
            }
            Operation[k] = can_1; //write operation type in Operation array
            Operation[k+1] = can_2;
            k+=2;
            if(can_3 >= 4){ //change dl register to original value
                Operation[k] = 0xb2;
                Operation[k+1] = 0x02;
                k+=2;
            }
        }
        else{ //add or sub or imul case
            Operation[k] = can_1; //put operation and value
            Operation[k+1] = can_2;
            Operation[k+2] = can_3;
            k+=3;
        }
    }
    Operation[k] = compiled_code[f]; //put no add/sub/imul/div value
    k++;
    f++;
    can_1=0; //empty current function
    can_2=0;
    can_3=0;
}
else{ //add/sub/imul/div but that is not current running-function
```

만약 add/sub/imul/div가 아닌 함수가 들어올 경우 최적화가 필요가 없다. 우선, 기존에 최적화를 위해 저장해두던 function이 있을 경우 해당 값들을 한 Operation으로서 업로드한다. 이때, div의 경우 2번 이상 나와 dl register의 고유값의 제공보다 큰 값이 되었을 경우, div가 dl register를 사용해 나눗셈을 진행하여야 한다는 조건을 만족하기 위해 해당 위치에 접근하여 값을 can_3(value)로 바꾼 후 div를 진행한 다음 다시 고유값인 2로 dl register의 값을 변경하였다. 이후 들어온 operation을 operation에 넣고, 최적화 중인 function이 없어졌으니 can_1, 2, 3를 0으로 초기화한다.

```

else{
    //add/sub/imul/div but that is not current running-function
    if(can_1 != 0x00 && can_2 != 0x00){
        //current function is not empty
        if(can_1 == 0xf6 && can_2 == 0xf2){//div case
            if(can_3 >= 4){
                Operation[k] = 0xb2;
                Operation[k+1] = can_3;
                k+=2;
            }
            Operation[k] = can_1;
            Operation[k+1] = can_2;
            k+=2;
            if(can_3 >= 4){
                Operation[k] = 0xb2;
                Operation[k+1] = 0x02;
                k+=2;
            }
        }
        else{ //add sub imul case
            Operation[k] = can_1;
            Operation[k+1] = can_2;
            Operation[k+2] = can_3;
            k+=3;
        }
    }

    can_1 = compiled_code[f]; //update current function
    can_2 = compiled_code[f+1];
    if(can_1 == 0xf6 && can_2 == 0xf2){//when div, we need to set can_3 to dl register
        can_3=2;
        f+=2;
    }
    else{
        can_3=compiled_code[f+2];
        f+=3;
    }
}
}while(compiled_code[f] != 0xC3); //check asm array's end

```

만약 add/sub/imul/div 함수가 읽혀졌지만, 현재 최적화 진행중인 function이 아닌 경우 다른 함수가 들어왔을 때와 마찬가지로 최적화 중이던 Operation의 정보를 올림과 동시에 can_3에 내장된 값을 통해 하나의 Operation으로 동작이 가능하게 하였다. (div의 경우 앞과 마찬가지로 mov를 이용해 dl register에 접근한다.) 이후 새로 들어온 Operation에 대해 최적화를 진행하기 위해 can_1, can_2, can_3에 이 값들을 받게 된다.

위와 같은 진행을 0xC3가 감지되어 코드의 끝임을 인지할 때 까지 한다.

```

if(can_1 != 0x00 && can_2 != 0x00){
    //before asm array is end, we need to update current function
    if(can_1 == 0xf6 && can_2 == 0xf2){//when div case
        if(can_3 >= 4){
            //div time is over two
            Operation[k] = 0xb2; //change dl register value to can_3
            Operation[k+1] = can_3;
            k+=2;
        }
        Operation[k] = can_1; //do div
        Operation[k+1] = can_2;
        k+=2;
        if(can_3 >= 4){
            //change dl register value to original value
            Operation[k] = 0xb2;
            Operation[k+1] = 0x02;
            k+=2;
        }
    }
    else{
        //add sub imul case
        //put Operation and value
        Operation[k] = can_1;
        Operation[k+1] = can_2;
        Operation[k+2] = can_3;
        k+=3;
    }
}
Operation[k]=compiled_code[f]; //put end of asm code(0xC3)

mprotect(Operation, pagesize, PROT_NONE);
mprotect(Operation, pagesize, PROT_READ | PROT_EXEC); //change authority to exec
msync(Operation, pagesize, MS_ASYNC); //synchronize map update
return Operation; //return to use function

```

코드가 끝났을 때 최적화 중인 Operation이 있다면 이 또한 받아야 하기에 위에서 처리해 준 방식과 동일하게 Operation에 넣어준다. 이후 코드의 끝을 의미하는 0xC3또한 Operation내에 넣어준다. 이후 Operation은 실행이 가능해야 하기에 mprotect를 이용, Read-Write였던 Operation의 권한을 Read Exec로 변경해주고 해당 변경을 msync를 통해 업데이트 해주었다. 이후 return을 통해 함수 포인터에 해당 Operation 배열을 넣어 실행이 가능하게 해준다.

최적화 방식은 다음과 같다. Add a, add b, add c => add (a+b+c)

sub a, sub b, sub c => sub (a+b+c)

imul a, imul b, imul c => imul (a*b*c)

div n, div n, div n => div (n*n*n) * n은 dl register에 입력된 고유값

```

#else //when make
pagesize = getpagesize(); //make page size

Operation = (uint8_t *)mmap(0, pagesize, PROT_READ | PROT_WRITE, MAP_ANONYMOUS | MAP_SHARED, -1, 0);
//put memory Operation (read write)
k=0;
do{
    Operation[k]=compiled_code[k];
    //put all raw asm Operation in Operation array
}while(k++<len-1);
mprotect(Operation, pagesize, PROT_NONE);
mprotect(Operation, pagesize, PROT_READ | PROT_EXEC); //change authority to exec
msync(Operation, pagesize, MS_ASYNC); //synchronize map update
return Operation; //return to use function
#endif

```

만약 make dynamic이 아닌 make의 경우(#else~#endif)는 Operation의 값을 변경해야 하는 건 동일하니 read-write 권한으로 mmap을 진행한 후 compiled_code 내 값들을 그대로 Operation 내로 옮긴다. 이후 mprotect로 실행이 가능하도록 권한을 변경하고 msync를 통해 이를 업데이트 한다. 이후 Operation을 함수포인터에 return한다.

```

os2018202074@ubuntu: ~/assign_4/4-2
os2018202074@ubuntu:~/assign_4/4-2$ sync
os2018202074@ubuntu:~/assign_4/4-2$ echo 3 | sudo tee /proc/sys/vm/drop_caches
[sudo] password for os2018202074:
3
os2018202074@ubuntu:~/assign_4/4-2$ gcc -o test2 D_recompile_test.c
os2018202074@ubuntu:~/assign_4/4-2$ ./test2
Data was filled to shared memory.
os2018202074@ubuntu:~/assign_4/4-2$ make
cc -o drecompile D_recompile.c
os2018202074@ubuntu:~/assign_4/4-2$ ./drecompile

```

우선 위와 같이 실행하여 최적화를 안한 경우(make) 결과를 확인하였다.

```

os2018202074@ubuntu:~/assign_4/4-2$ ./drecompile
3
0(th) run time: 810.000000
3
1(th) run time: 1060.000000
3
2(th) run time: 1250.000000
3
47(th) run time: 850.000000
3
48(th) run time: 1000.000000
3
49(th) run time: 920.000000

total run time: 46580.000000
average run time: 931.600000
result: 15

```

위 같은 결과를 내보냈다.

처리 결과는 여러 번 돌려봐도 15가 나왔으며 시간에 대하여는 표로 나타내면 다음과 같다.(위 사진과 다른 회차)

SET	0	1	2	3	4	5	6	7
TIME	850	1700	840	920	840	850	880	1000
SET	8	9	10	11	12	13	14	15
TIME	840	840	850	850	850	890	850	850
SET	16	17	18	19	20	21	22	23
TIME	1230	1030	850	850	850	850	850	1390

SET	24	25	26	27	28	29	30	31
TIME	850	920	850	1000	850	850	850	950
SET	32	33	34	35	36	37	38	39
TIME	850	850	920	930	850	890	1000	1100
SET	40	41	42	43	44	45	46	47
TIME	850	850	1550	1000	940	1190	1170	850
SET	48	49			Total		average	
TIME	1000	920			47430(nano sec)		948.6(nano sec)	

```

os2018202074@ubuntu:~/assign_4/4-2$ sync
os2018202074@ubuntu:~/assign_4/4-2$ echo 3 | sudo tee /proc/sys/vm/drop_caches
3
os2018202074@ubuntu:~/assign_4/4-2$ gcc -o test2 D_recompile_test.c
os2018202074@ubuntu:~/assign_4/4-2$ ./test2
Data was filled to shared memory.
os2018202074@ubuntu:~/assign_4/4-2$ make dynamic
cc -Ddynamic -o drecompile D_recompile.c
os2018202074@ubuntu:~/assign_4/4-2$ ./drecompile

```

이번엔 최적화를 진행한 상태의 결과이다.

```

os2018202074@ubuntu:~/assign_4/4-2$ ./drecompile
3
0(th) run time: 890.000000
3
1(th) run time: 600.000000
3
2(th) run time: 490.000000
3
47(th) run time: 520.000000
3
48(th) run time: 530.000000
3
49(th) run time: 580.000000

total run time: 26760.000000
average run time: 535.200000
result: 15
os2018202074@ubuntu:~/assign_4/4-2$

```

Function의 계산 결과는 동일하되(result) 총, 평균 run time이 감소한 것을 확인할 수 있었다.

SET	0	1	2	3	4	5	6	7
TIME	890	600	490	420	500	420	470	490
SET	8	9	10	11	12	13	14	15
TIME	550	470	760	590	540	480	490	530
SET	16	17	18	19	20	21	22	23
TIME	580	490	470	470	550	490	560	640
SET	24	25	26	27	28	29	30	31
TIME	400	550	530	540	480	500	540	490
SET	32	33	34	35	36	37	38	39
TIME	520	550	540	720	590	490	580	560
SET	40	41	42	43	44	45	46	47
TIME	530	470	580	580	550	490	490	520
SET	48	49			Total		average	
TIME	530	580			26760(nano sec)		535.2(nano sec)	

이후 양식에 맞추기 위해 위와 같이 코드를 변경해 주었다.

```
int result=0;
int subresult=0;

//for(int l=0;l<50;l++){
    system("sync");
    system("echo 3 | sudo tee /proc/sys/vm/drop_caches");
    clock_gettime(CLOCK_MONOTONIC, &startTime); //start of time
    result=func(1);
    clock_gettime(CLOCK_MONOTONIC, &endTime); //end of time
    subdiffTime=1000000000 * (endTime.tv_sec - startTime.tv_sec)+(endTime.tv_nsec-
startTime.tv_nsec);
    //    printf("%d(th) run time: %lf\n", l,(double)subdiffTime); //print time(nano-sec)
    //    diffTime += subdiffTime;
    //}
    //printf("\ntotal run time: %lf\n", (double)diffTime); //print time(nano-sec)
    //printf("average run time: %lf\n", (double)diffTime/50); //print time(nano-sec)
    //printf("result: %d\n", result); //check asm right

    printf("total execution time: %.8lf sec\n", (double)subdiffTime/1000000000); //print time(nano-sec)

drecompile_exit();
sharedmem_exit();
return 0;
}
```

```
os2018202074@ubuntu:~/assign_4/4-2$ sync
os2018202074@ubuntu:~/assign_4/4-2$ echo 3 | sudo tee /proc/sys/vm/drop_caches
3
os2018202074@ubuntu:~/assign_4/4-2$ gcc -o test2 D_recompile_test.c
os2018202074@ubuntu:~/assign_4/4-2$ ./test2
Data was filled to shared memory.
os2018202074@ubuntu:~/assign_4/4-2$ make
cc -o drecompile D_recompile.c
os2018202074@ubuntu:~/assign_4/4-2$ ./drecompile
3
total execution time: 0.00000075 sec
os2018202074@ubuntu:~/assign_4/4-2$
```

```
os2018202074@ubuntu:~/assign_4/4-2$ sync
os2018202074@ubuntu:~/assign_4/4-2$ echo 3 | sudo tee /proc/sys/vm/drop_caches
3
os2018202074@ubuntu:~/assign_4/4-2$ gcc -o test2 D_recompile_test.c
os2018202074@ubuntu:~/assign_4/4-2$ ./test2
Data was filled to shared memory.
os2018202074@ubuntu:~/assign_4/4-2$ make dynamic
cc -Ddynamic -o drecompile D_recompile.c
os2018202074@ubuntu:~/assign_4/4-2$ ./drecompile
3
total execution time: 0.00000044 sec
os2018202074@ubuntu:~/assign_4/4-2$
```

고찰

이번 과제를 통해 이전과제를 하며 배웠던 wrapping을 포함 clock_gettime을 통한 시간 측정에 대한 내용도 복습할 수 있었고, 추가적으로 어셈블리를 c환경에서 어떤 식으로 다룬다는 생각을 못했었는데, 함수 포인터를 사용해 이를 받아올 수 있다는 점이 신기했다. 또한 위 과정을 통해 shared memory에 대해 어떻게 대응하여야 하는 지 알 수 있었으며 mmap을 활용해 memory를 접근시키는 방법에 대해서도 실습을 통해 이해하는 것이 가능했다.

아쉬운 점이 있다면 div의 경우 register가 지정되어 있어 다른 register를 사용할 수 있다면 mov를 한번만 사용해도 되는 것을 2번씩 사용하여 최적화에 아쉬움을 남겼다.

Reference

강의자료 참고

22-2_OSLab_11_Shared_memory.pdf

22-2_OSLab_12_Memory_management.pdf

Task_struct로부터 파일명 파일 경로 출력하기 : <https://beausty23.tistory.com/109>

리눅스 커널의 mm_struct / vm_area_struct 구조체 : <https://showx123.tistory.com/92>

함수 포인터: <https://boycoding.tistory.com/233>

2022-2_OSLab_Assignment_4.pdf