운영체제

Assignment4

한준호
2018741035
최상호 교수님

Assignment 4-1

- Introduction

2차 과제에서 작성한 ftracae 시스템 콜(336번)을 'file_varea' 함수로 wrapping 하여, PID를 바탕으로 프로세스 정보를 출력하는 Module을 작성한다.
출력하는 프로세스 정보: 프로세스의 이름과 pid, 정보가 위치하는 가상 메모리 주소, 프로세스의 데이터 주소, 코드 주소, 힙 주소, 정보의 원본 파일의 전체 경로

- Conclusion & Analysis

• 'file_varea.c'

```c
#include <linux/module.h>
#include <linux/sched.h>
#include <linux/init_task.h>

#include <linux/highmem.h>
#include <linux/kallsyms.h> /* kallstms_lookup_name() */
#include <linux/syscalls.h> /* __SYSCALL_DFINRx() */
#include <asm/syscall_wrapper.h> /* __SYSCALL_DFINRx() */
#include <linux/sched/mm.h>
#include <linux/unistd.h>
#include <linux/mm_types.h>

#define __NR_ftrace 336
```

```c
void **syscall_table;

unsigned int pid_n = 0;

asmlinkage int(*real_ftrace)(pid_t);
asmlinkage int file_varea(pid_t pid)
{
        struct task_struct *t;
        struct mm_struct *mm;
        struct vm_area_struct *vm;
        int mm_count =0;

        t = pid_task(find_vpid(pid), PIDTYPE_PID);
        mm = get_task_mm(t);
        down_read(&mm->mmap_sem);
        vm = mm->mmap;
        up_read(&mm->mmap_sem);

        printk(KERN_INFO "######## Loaded files of a process '%s(%d)' in VM ########\n", t->comm, pid);
        //printk(KERN_INFO "name[%s]\n", t->comm);

        for(mm_count = mm->map_count; 0<mm_count; mm_count--)
        {
                char* file_path;
                char buf[100];
                struct file* file;

                file = vm->vm_file;
                file_path = d_path(&file->f_path, buf, 100);

                printk(KERN_INFO "mem[%081x - %081x] ", vm->vm_start, vm->vm_end);
                printk(KERN_INFO "code[%081x - %081x] ", mm->start_code, mm->end_code);
                printk(KERN_INFO "data[%081x - %081x] ", mm->start_data, mm->end_data);
                printk(KERN_INFO "heap[%081x - %081x] ", mm->start_brk, mm->brk);
                printk(KERN_INFO "%s\n", file_path);

                vm = vm->vm_next;
        }
        mmput(mm);

        return 0;
}
```

```
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 file_varea_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] = file_varea;

        return 0;
}

static void file_varea_exit(void)
{
        printk(KERN_INFO "################################################\n");
        syscall_table[__NR_ftrace] = real_ftrace;
        make_ro(syscall_table);
}

module_init(file_varea_init);
module_exit(file_varea_exit);
MODULE_LICENSE("GPL");
                                                              93,1          Bot
```

2차 과제에서 만든 ftrace 시스템콜을 hijack 하여 'file_varea' 함수로 대체하기 위해 'file_varea.c' 를 작성하였고, wrapping 방법으로는 'asmlinkage' 를 사용하였다. 모듈이 추가될 때 system call table에 읽기 및 쓰기 권한을 부여하고, 모듈이 제거될 때 읽기 및 쓰기 권한을 회수할 수 있도록 함수를 작성하였다.

```
struct task_struct *t;
struct mm_struct *mm;
struct vm_area_struct *vm;
int mm_count =0;
```

file_varea.c에서 PID를 바탕으로 프로세스의 정보를 출력하기 위해 프로세스의 정보를 담고있는 구조체를 선언해주었다.

Linux Kernel 홈페이지(https://docs.huihoo.com/doxygen/linux/kernel/3.7/structtask__struct.html)에서 task_struct, mm_struct, vm_area_struct 구조체의 정보를 확인하였다.

task_struct 구조체가 mm_struct 구조체를 포함하고, mm_struct 구조체가 vm_area_struct 구조체를 포함하고 있는 구조이다.

• task_struct

task_struct Struct Reference

```
#include <sched.h>
```

Data Fields

volatile long	state
void *	stack
atomic_t	usage
unsigned int	flags
unsigned int	ptrace
int	on_rq
int	prio
int	static_prio
int	normal_prio
unsigned int	rt_priority
struct sched_class *	sched_class
struct sched_entity	se
struct sched_rt_entity	rt
unsigned char	fpu_counter
unsigned int	policy
int	nr_cpus_allowed
cpumask_t	cpus_allowed
struct list_head	tasks
struct mm_struct *	mm
struct mm_struct *	active_mm
int	exit_state
int	exit_code
int	exit_signal
int	pdeath_signal
unsigned int	jobctl
unsigned int	personality
unsigned	did_exec:1
unsigned	in_execve:1
unsigned	in_iowait:1
unsigned	no_new_privs:1
unsigned	sched_reset_on_fork:1
unsigned	sched_contributes_to_load:1
pid_t	pid
pid_t	tgid
struct task_struct __rcu *	real_parent
struct task_struct __rcu *	parent
struct list_head	children
struct list_head	sibling
struct task_struct *	group_leader
struct list_head	ptraced
struct list_head	ptrace_entry
struct pid_link	pids [PIDTYPE_MAX]
struct list_head	thread_group
struct completion *	vfork_done
int __user *	set_child_tid
int __user *	clear_child_tid
cputime_t	utime
cputime_t	stime
cputime_t	utimescaled
cputime_t	stimescaled
cputime_t	gtime
cputime_t	prev_utime
cputime_t	prev_stime
unsigned long	nvcsw
unsigned long	nivcsw
struct timespec	start_time
struct timespec	real_start_time
unsigned long	min_flt
unsigned long	maj_flt
struct task_cputime	cputime_expires
struct list_head	cpu_timers [3]
struct cred __rcu *	real_cred
struct cred __rcu *	cred
char	comm [TASK_COMM_LEN]
int	link_count
int	total_link_count
struct thread_struct	thread
struct fs_struct *	fs
struct files_struct *	files
struct nsproxy *	nsproxy
struct signal_struct *	signal
struct sighand_struct *	sighand
sigset_t	blocked
sigset_t	real_blocked
sigset_t	saved_sigmask
struct sigpending	pending
unsigned long	sas_ss_sp
size_t	sas_ss_size
int(*	notifier)(void *priv)
void *	notifier_data
sigset_t *	notifier_mask
struct callback_head *	task_works
struct audit_context *	audit_context
struct seccomp	seccomp
u32	parent_exec_id
u32	self_exec_id
spinlock_t	alloc_lock
raw_spinlock_t	pi_lock
void *	journal_info
struct bio_list *	bio_list
struct reclaim_state *	reclaim_state
struct backing_dev_info *	backing_dev_info
struct io_context *	io_context
unsigned long	ptrace_message
siginfo_t *	last_siginfo
struct task_io_accounting	ioac
struct rcu_head	rcu
struct pipe_inode_info *	splice_pipe
struct page_frag	task_frag
int	nr_dirtied
int	nr_dirtied_pause
unsigned long	dirty_paused_when
unsigned long	timer_slack_ns
unsigned long	default_timer_slack_ns

• mm_struct

mm_struct Struct Reference

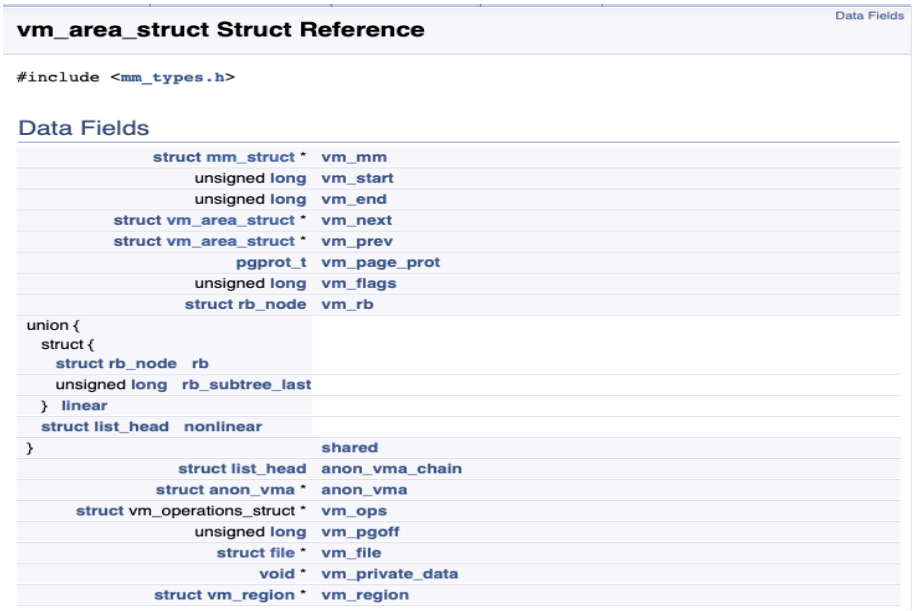
```
#include <mm_types.h>
```

Data Fields

struct vm_area_struct *	mmap
struct rb_root	mm_rb
struct vm_area_struct *	mmap_cache
unsigned long	mmap_base
unsigned long	task_size
unsigned long	cached_hole_size
unsigned long	free_area_cache
pgd_t *	pgd
atomic_t	mm_users
atomic_t	mm_count
int	map_count
spinlock_t	page_table_lock
struct rw_semaphore	mmap_sem
struct list_head	mmlist
unsigned long	hiwater_rss
unsigned long	hiwater_vm
unsigned long	total_vm
unsigned long	locked_vm
unsigned long	pinned_vm
unsigned long	shared_vm
unsigned long	exec_vm
unsigned long	stack_vm
unsigned long	def_flags
unsigned long	nr_ptes
unsigned long	start_code
unsigned long	end_code
unsigned long	start_data
unsigned long	end_data
unsigned long	start_brk
unsigned long	brk
unsigned long	start_stack
unsigned long	arg_start
unsigned long	arg_end
unsigned long	env_start
unsigned long	env_end
unsigned long	saved_auxv [AT_VECTOR_SIZE]
struct mm_rss_stat	rss_stat
struct linux_binfmt *	binfmt
cpumask_var_t	cpu_vm_mask_var
mm_context_t	context
unsigned long	flags
struct core_state *	core_state
struct file *	exe_file
struct uprobes_state	uprobes_state

- mm_users : mm_struct의 주소 공간을 사용중인 task를 나타낸다. 만약 하나의 프로세스에 대해 메모리를 공유하는 스레드가 3개라면, mm_users는 3이 된다.
- mm_count : mm_struct의 참조 카운트. 0이 되면 mm_struct가 free된다.
- mmap : 가상 메모리 영역의 리스트. vm_area_struct에 대해서는 아래에서 다룬다.
- mm_rb : 마찬가지로 가상 메모리 영역을 저장한다. mmap의 원소와 완전히 동일하지만 red-black tree 구조이다. mmap은 VMA를 순회할때 사용되고, mm_rb는 VMA를 검색할 때 사용된다.
- pgd : 가상메모리 -> 물리메로리 매핑에 사용되는 페이지 테이블의 주소.
- start_code / end_code : code영역의 시작 주소와 끝 주소. 유사하게, data영역이나(start_data) heap영역(start_brk)도 다른 멤버로 선언되어있다.

• vm_area_struct



vm_area_struct Struct Reference Data Fields

```
#include <mm_types.h>
```

Data Fields

struct mm_struct *	vm_mm
unsigned long	vm_start
unsigned long	vm_end
struct vm_area_struct *	vm_next
struct vm_area_struct *	vm_prev
pgprot_t	vm_page_prot
unsigned long	vm_flags
struct rb_node	vm_rb
union { struct { struct rb_node rb; unsigned long rb_subtree_last; } linear; struct list_head nonlinear; }	shared
struct list_head	anon_vma_chain
struct anon_vma *	anon_vma
struct vm_operations_struct *	vm_ops
unsigned long	vm_pgoff
struct file *	vm_file
void *	vm_private_data
struct vm_region *	vm_region

- vm_start, vm_end : VMA 영역의 시작 주소와 끝 주소. vm_end - vm_start가 해당 VMA의 사이즈를 나타낸다.
- vm_prev, vm_next : 다음과 이전 VMA 구조체를 가르키는 포인터. 더블 링크드 리스트 형태로 되어있고, vm_start로 소팅되어 있다.
- vm_mm : 해당 VMA가 속해있는 mm_struct
- vm_flags : 해당 메모리 영역의 속성을 나타내는 플래그
- vm_ops : VMA 영역에 대한 opeartions들을 나타내는 function pointer들
- vm_file : mmap을 사용한 경우 해당하는 파일, null일 경우 이 VMA는 mmap으로 사용되고 있지 않음.

```
t = pid_task(find_vpid(pid), PIDTYPE_PID);
mm = get_task_mm(t);
down_read(&mm->mmap_sem);
vm = mm->mmap;
up_read(&mm->mmap_sem);
```

pid_task() 함수를 이용해 프로세스의 PID를 위에서 선언한 task_struct 구조체 t에 저장한다.

get_task_mm() 함수를 이용해 task_struct 구조체 t에 저장한 PID에 해당하는 정보를 위에서 선언한 mm_struct 구조체 mm에 연결한다.

down_read() 함수를 이용해 메모리 공간에 대한 읽기 잠금을 설정하고 mm_struct 구조체 mm에 해당하는 정보를 위에서 선언한 vm_area_struct 구조체 vm에 연결한다. 연결 후에, up_read() 함수를 이용해 메모리 세파포어(mmap_sem)에 대한 읽기 잠금을 해제한다.

```
printk(KERN_INFO "######## Loaded files of a process '%s(%d)' in VM ########\n", t->comm, pid);
//printk(KERN_INFO "name[%s]\n", t->comm);
```

프로세스의 이름과 pid를 출력한다. 이 때, task_struct 구조체 t의 comm에 프로세스 이름이 저장되어 있고 pid 변수에 프로세스의 PID가 저장되어있다.

```
for(mm_count = mm->map_count; 0<mm_count; mm_count--)
{
        char* file_path;
        char buf[100];
        struct file* file;

        file = vm->vm_file;
        file_path = d_path(&file->f_path, buf, 100);

        printk(KERN_INFO "mem[%081x - %081x] ", vm->vm_start, vm->vm_end);
        printk(KERN_INFO "code[%081x - %081x] ", mm->start_code, mm->end_code);
        printk(KERN_INFO "data[%081x - %081x] ", mm->start_data, mm->end_data);
        printk(KERN_INFO "heap[%081x - %081x] ", mm->start_brk, mm->brk);
        printk(KERN_INFO "%s\n", file_path);

        vm = vm->vm_next;
}
mmput(mm);

return 0;
```

vm_area_struct 구조체 vm에 할당된 가상 메모리 주소 범위가 저장되어 있는 vm_start, vm_end를 이용해 정보가 위치하는 가상 메모리 주소를 출력한다.

mm_struct 구조체 mm에 코드 주소의 범위가 저장되어 있는 start_code, end_code를 이용해 프로세스의 코드 주소를 출력한다.
mm_struct 구조체 mm에 데이터 주소의 범위가 저장되어 있는 start_data, end_data를 이용해 프로세스의 데이터 주소를 출력한다.

mm_struct 구조체 mm에 힙 주소의 범위가 저장되어 있는 start_brk, brk를 이용해 프로세스의 힙 주소를 출력한다.

file_varea.c에서 현재 프로세스의 task_struck를 이용해 test.c(assign4) 파일의 경로를 얻기 위해 d_path() 함수를 이용해 'file_path' 변수의 파일의 경로를 저장하고, 정보의 원본 파일의 전체 경로를 출력한다.

이 때, mm_struct 구조체 mm의 map_count와 위에서 선언한 mm_count 변수를 이용해, vm = vm->next로 넘겨가며 반복문으로 출력하여 모든 프로세스의 정보를 출력할 수 있도록 하였다.

```
/assignment4/4-1$ sudo insmod file_varea.ko
/assignment4/4-1$ ./assign4

/assignment4/4-1$ sudo rmmod file_varea.ko
/assignment4/4-1$ dmesg
```

```
######## Loaded files of a process 'assin4(45339)' in VM ########
mem[400000~401000] code[400000~40074c] data[600e10~601040] heap[1fe9000~1fe9000] /home/sslab/assign4/4_1/assin4
mem[600000~601000] code[400000~40074c] data[600e10~601040] heap[1fe9000~1fe9000] /home/sslab/assign4/4_1/assin4
mem[601000~602000] code[400000~40074c] data[600e10~601040] heap[1fe9000~1fe9000] /home/sslab/assign4/4_1/assin4
mem[7f4d3f41a000~7f4d3f5da000] code[400000~40074c] data[600e10~601040] heap[1fe9000~1fe9000] /lib/x86_64-linux-gnu/libc-2.23.so
mem[7f4d3f5da000~7f4d3f7da000] code[400000~40074c] data[600e10~601040] heap[1fe9000~1fe9000] /lib/x86_64-linux-gnu/libc-2.23.so
mem[7f4d3f7da000~7f4d3f7de000] code[400000~40074c] data[600e10~601040] heap[1fe9000~1fe9000] /lib/x86_64-linux-gnu/libc-2.23.so
mem[7f4d3f7de000~7f4d3f7e0000] code[400000~40074c] data[600e10~601040] heap[1fe9000~1fe9000] /lib/x86_64-linux-gnu/libc-2.23.so
mem[7f4d3f7e4000~7f4d3f80a000] code[400000~40074c] data[600e10~601040] heap[1fe9000~1fe9000] /lib/x86_64-linux-gnu/ld-2.23.so
mem[7f4d3fa09000~7f4d3fa0a000] code[400000~40074c] data[600e10~601040] heap[1fe9000~1fe9000] /lib/x86_64-linux-gnu/ld-2.23.so
mem[7f4d3fa0a000~7f4d3fa0b000] code[400000~40074c] data[600e10~601040] heap[1fe9000~1fe9000] /lib/x86_64-linux-gnu/ld-2.23.so
############################################################
```

• 'Makefile'

```
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
```

```
os2018741035@ubuntu:~/Downloads/linux-4.19.67/assignment4/4-1$ 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
```

'file_varea.c'와 테스트용 파일인 test.c 파일도 같이 컴파일 되도록 Makefile을 작성하였다. 컴파일 됐을 때, test.c의 실행 파일은 'assign4'로 설정하였다.

- 고찰

PID를 바탕으로 프로세스 정보를 출력하는 Module을 작성하는 과정에서 task_struck 구조체, mm_struct 구조체, vm_area_struct 구조체를 이용했는데 task_struct 구조체는 앞 과제에서 몇번 다뤄보았지만 mm_struct 구조체, vm_area_struct 구조체는 다뤄본 적이 없었기 때문에 구조체 내부에 어떤 구조로 구성되어 있는지 몰랐고 과제를 진행하는데 어려움을 느꼈다. Linux Kernel 홈페이지를 찾아 각 구조체에 대한 정보를 확인하였고 이를 통해 과제를 해결하였다.

처음에 이 과제를 진행할 때, ftrace 시스템 콜을 'file_varea' 함수로 wrapping 하는 과정에서 앞에 과제에서는 'asmlinkage' 를 사용했지만, 이번 과제에서는 강의자료 예시 코드를 참고하여 '__SYSCALL_DEFINEx'을 사용해 wrapping을 진행해보려고 했지만, 구현 과정에서 컴파일시 커널 버전이 맞지 않다는 오류가 발생했고, 결국 'asmlinkage' 를 사용해서 wrapping을 진행하였다.

과제를 구현하는 과정에서, 과제 관련 정보를 찾는데 커널 버전에 따라 사용하는 함수가 변경되어 있는 경우가 있어서 사용하는 커널 버전에 맞는 함수를 찾는데 어려움을 느꼈다.

- Reference

• Linux Kernel 홈페이지, https://docs.huihoo.com/doxygen/linux/kernel/3.7/structtask__struct.html

• [Linux] task_struct로부터 파일명, 파일 경로 출력하기, https://beausty23.tistory.com/109

• 프로세스 메모리 분석 모듈. https://m.blog.naver.com/PostView.nhn?blogId=hsmnim&logNo=30108957173&proxyReferer=https://www.google.com/

Assignment 4-2

- Introduction

동적 재컴파일에 대해 학습하여 Memory를 공유하고 공유된 Memory에서 컴파일 된 코드에 접근하여 시스템이 실행 중에 프로그램의 일부를 재컴파일 하는 방법을 학습한다. 이 때, objdump 사용법을 학습하고 objdump를 이용해 dump 뜬 파일로부터 문제 해결의 실마리를 찾고 해결하는 방법을 학습한다. 동적 재컴파일을 진행하기 전, 즉 최적화하기 전 결과와 최적화를 한 결과를 비교해본다.

- Conclusion & Analysis

• Dynamic Recompilation

동적 재컴파일은 시스템이 실행 중에 프로그램의 일부를 재컴파일할 수 있는 일부 에뮬레이터 및 가상 머신의 기능이다.

실행 중에 컴파일함으로써 시스템은 프로그램의 런타임 환경을 반영하도록 생성된 코드를 조정할 수 있으며 잠재적으로 정보를 활용하여 보다 효율적인 코드를 생성할 수 있다.

• objdump

objdump는 GNU 바이너리 유틸리티의 일부로서, 라이브러리, 컴파일된 오브젝트 모듈, 공유 오브젝트 파일, 독립 실행파일등의 바이너리 파일들의 정보를 보여주는 프로그램이다. objdump는 ELF 파일을 어셈블리어로 보여주는 역어셈블러로 사용될 수 있다.

• objdump 사용법

objdump [-d] [-S] [-l] [obj 파일명] > [output 파일명]

'-d' 옵션은 disassemble, 즉 어셈블리어 코드 목록을 생성한다.
'-S' 옵션은 소스코드를 최대한 보여주는 옵션이다. optimization 된 object file의 경우 소스코드가 모두 보이지 않을 수 있다.
'-l' 옵션은 소스코드에서의 line을 보여주는 옵션이다.

'> [output 파일명]' 을 하게 되면 파일명으로 export 하게된다. 이를 생략하면 화면에 즉시 출력된다.

• 'D_recompile.c'

```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>

#define KEY_NUM 1234
#define MEM_SIZE 4096

int shmid;
uint8_t* Operation;
uint8_t* compiled_code;

void sharedmem_init(); // °øÀ¯ ¸Þ¸ð¸®¿¡ Á¢±Ù
void sharedmem_exit();
void drecompile_init(); // memory mapping ¼ÂÆÃ
void drecompile_exit();
void* drecompile(uint8_t *func); //ÃÖÀûÈ ÇÏ´Â ºÎºÐ

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

        sharedmem_init();
        drecompile_init();

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

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

```c
void sharedmem_init()
{
        if((shmid =shmget((key_t)KEY_NUM, MEM_SIZE, IPC_CREAT|IPC_EXCL|0666)) == -1)
        {
                if(shmid==-1)
                {
                        perror("Shared memory init failed");
                        return 1;
                }
                else
                {
                        if(shmctl(shmid, IPC_RMID, 0)==-1)
                        {
                                perror("Shmctl failed");
                                return 1;
                        }

                        shmid = shmget((key_t)KEY_NUM, MEM_SIZE, IPC_CREAT|0666);

                        if(shmid==-1)
                        {
                                perror("Shared memory init failed");
                                return 1;
                        }
                }

        }
        return 0;
}

void sharedmem_exit()
{
        shmid = shmget((key_t)3846, sizeof(SHM_INFOS)*SHM_INFO_COUNT, 0666|IPC_CREAT);
        if (shmid == -1)
        {
                perror("shmget failed")
                exit(0);
        }

        if (-1 == shmctl(shmid, IPC_RMID, 0))
        {
                return -1;
        }

        return 0;
}
```

```
void drecompile_init(uint8_t *func)
{
        string code = System.ID.File.ReadALLTEXT("./D_recompile_test.c");

        CodeDomProvider codeDom = CodeDomProvider.CreateProvider("C");

        CompilerParameters cparams = new CompilerParameters();
        cparams.GenerateExecutable = true;
        cparams.OutputAssembly = "TEST.EXT";

        CompilerResults results = codeDom.CompileAssemblyFromSource(cparams, code);

        if(results.Errors.Count > 0)
        {
                foreach (var err in results.Errors)
                {
                        Console.WriteLine(err.ToString());
                }
                return;
        }
        Process.Start("TEST.EXE");
}

void drecompile_exit()
{
}

void* drecompile(uint8_t* func)
{
}
```

〈Shared Memory〉

• shmget

함수 원형: int shmget(key_t key, int size, int shmflg)

- 커널에 공유 메모리 공간을 요청하기 위해 사용하는 시스템 호출 함수
- KEY값은 고유의 공유 메모리임을 나타낸다
- Argument (KEY, MEMORY_MAX_SIZE, 접근권한 | 생성방식)

• shmmat

함수 원형: void *shmat(int shmid, const void *shmaddr, int shmflg)

- 공유 메모리 공간을 생성한 이후, 공유메모리에 접근할 수 있는 int형의 "식별자"를 얻는다
- 공유 메모리를 사용하기 위해 얻은 식별자를 이용하여 현재 프로세스가 공유 메모리에 접근할 수 있도록 연결하는 작업
- Argument (식별자, 메모리가 붙을 주소 (0을 사용할 경우 커널이 메모리가 붙을 주소를 명시), 읽기/쓰기 모드)

• shmdt

함수 원형: int shmdt(const void *shmaddr)

- 프로세스가 더이상 공유 메모리를 사용하지 않을 경우 프로세스와 공유 메모리를 분리시키는 작업

- 해당 시스템 호출 함수는 현재 프로세스와 공유 메모리를 분리시킬 뿐, 공유 메모리의 공간을 삭제하지 않는다
- shmdt 가 성공적으로 수행되면 커널은 shmid_ds 의 내용을 갱신, 즉 shm_dtime, shm_lpid, shm_nattch 등의 내용을 갱신 하는데, shm_dtime 는 가장 최근에 dettach (즉 shmdt 를 사용한)된 시간, shm_lpid 는 호출한 프로세세의 PID, shm_nattch 는 현재 공유 메모리를 사용하는 (shmat 를 이용해서 공유 메모리에 붙어 있는) 프로세스의 수를 돌려준다.

• shmctl
함수 원형: int shmctl(int shmid, int cmd, struct shmid_ds *buf)
- 공유 메모리를 제어하기 위해 사용
- shmid_ds 구조체를 직접 제어함으로써, 해당 공유 메모리에 대한 소유자, 그룹 등의 허가권을 변경하거나, 공유 메모리 삭제, 공유 메모리의 잠금을 설정하거나 헤제하는 작업
- Option
IPC_STAT : 공유 메모리 공간에 관한 정보를 가져오기 위해서 사용된다. 정보는 buf 에 저장된다.
IPC_SET : 공유 메모리 공간에 대한 사용자권한 변경을 위해서 사용된다. 사용자 권한 변경을 위해서는 슈퍼유저 혹은 사용자 권한을 가지고 있어야 한다.
IPC_RMID : 공유 메모리 공간을 삭제하기 위해서 사용된다. 이 명령을 사용한다고 해서 곧바로 사용되는건 아니며, 더 이상 공유 메모리 공간을 사용하는 프로세스가 없을 때, 즉 shm_nattch 가 0일때 까지 기다렸다가 삭제된다.

〈drecompile (구현하지 못함)〉

Recompile할 코드를 가져온 뒤, 컴파일 언어를 'c' 로 설정해주었다.

컴파일러의 파라미터 옵션을 지정하고 소스코드를 컴파일해서 EXE를 생성 한 후, 컴파일 후 결과를 저장해주었다.

컴파일 에러를 확인하고, 에러가 있다면 에러를 출력하도록 하였다.

컴파일 결과(EXE)를 실행해주었다.

• '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;"
        "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;"
        "sub %%eax, 3;"
        "sub %%eax, 1;"
        "sub %%eax, 1;"
        "sub %%eax, 1;"
        "sub %%eax, 3;"
        "sub %%eax, 1;"
        "sub %%eax, 1;"
"D_recompile_test.c" [dos] 1095L, 25134C
```

Dynamic Recompilation을 하는 test용 파일은 강의자료에 올려주신 파일을 그대로 사용하였다.

• 'Makefile'

```
EXEC = D_recompile
CC = gcc

default:
        $(CC) -o drecompile D_recompile.c
dynamic:
        $(CC) -Dynamic -o drecompile D_recompile.c
clean:
        rm -rf D_recompile $(EXEC)
~
```

• 결과 분석 (오류 나옴)

```
os2018741035@ubuntu:~/assignment4/4-2$ 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;
         ^
```

- 고찰

동적 재컴파일에 대해 학습하여 Memory를 공유하고 공유된 Memory에서 컴파일 된 코드에 접근하여 시스템이 실행 중에 프로그램의 일부를 재컴파일 하는 방법을 학습하였지만 과제 구현과 관련하여 많은 정보를 찾지 못했고 과제 구현에 어려움을 느꼈다.

결국, 이 과정에서 objdump 사용법은 학습 하였지만, 끝까지 과제를 완벽하게 구현하지 못했기 때문에 동적 재컴파일을 진행하기 전, 즉 최적화하기 전 결과와 최적화를 한 결과를 비교 해보지 못했다.

Memory를 구현하는 방법에 대해서는 학습 하였지만, Dynamic Recompile을 하는 법에 대해서는 학습이 부족했고, 이 부분에서 구현에 어려움을 느꼈다.

- Reference

• objdump 사용법, https://log1.tistory.com/3

• Shared Memory 정리 및 예제, https://coding-chobo.tistory.com/16

• 예제로 배우는 C# 프로그래밍, https://www.csharpstudy.com/Practical/Prac-dynamic-compile.aspx