Assignment3

운영체제

Files located in VM Area

제출일: 11월 25일 일요일

담당 교수: 김태석

학 번: 2015722025

학 과: 컴퓨터정보공학부

이 름: 정용훈

1. Introduction

3차 과제는 한 프로세스에 대하여 정보가 위치하는 가상 메모리 주소, 프로세스의 데이터 주소, 코드 주소, heap 주소, 정보 원본 파일의 전체 경로를 출력하는 System call을 작성하는 것이다. 그럼으로써 모듈에 대한 개념을 다시 한번 익히고, 메모리 관리를 위한 자료 구조와 관련 함수에 대한 사용 능력을 배양하는데 목적을 둔다.

2. Conclusion (& 과제 요구사항)

(1) 이전 과제에서 사용한 add() 시스템 콜에 wrapping 할 시스템 콜 작성

```c
#include <stdio.h> //library
#include <sys/types.h>
#include <unistd.h>
#define __NR_add 549 //number of system call

int main()
{
        int a,b;
        a=7;
        b=4;
        #ifdef WRAPPING
                //The function is used when work wrapping
                printf("%d add %d = %ld\n", a, b, syscall(__NR_add, getpid()));
        #else
                //The function is base condition
                printf("%d add %d = %ld\n", a, b, syscall(__NR_add, a, b));
        #endif
        return 0;
}
```

전 과제에서 등록해놓은 add system call을 test하기 위한 source file이다. 변경 사항은 일반 적인 compile로 동작하는 함수는 #else에 의한 함수이며, 이는 우리가 미리 등록해놓은 덧셈을 동작하는 함수이다. 전에 등록했던 함수를 사용하기 위해선 아래와 같은 명령을 사용하여 실행 파일을 생성하면 된다.

```
os2015722025@ubuntu:~/Desktop/OS_3$ make test
gcc -o test test.c
os2015722025@ubuntu:~/Desktop/OS_3$ ./test
7 add 4 = 11
```

정상적으로 code가 동작하는 것을 확인할 수 있다. 반면, 이번 과제를 수행하기 위해 현재 process의 ID를 인자로 하는 wrapping된 add 함수를 실행하기 위해서는 전처리 #ifdef로 정의된 code를 compile해야 하는데 해당 code를 compile하기 위해서는 다음과 같이 명령을 입력해주면 된다.

```
os2015722025@ubuntu:~/Desktop/OS_3$ make testwrapping
gcc -o test test.c -DWRAPPING
os2015722025@ubuntu:~/Desktop/OS_3$ ./test
7 add 4 = 16080
```

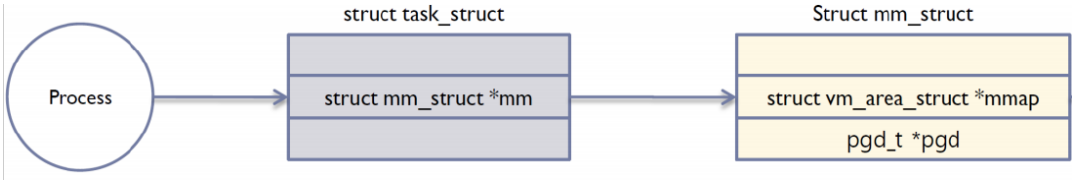
다음과 같이 현재 process ID가 출력되는 것을 확인할 수 있다. (정확한 ID값은 아님)

```c
#include <linux/module.h>
#include <linux/highmem.h>
#include <linux/kallsyms.h>
#include <linux/syscalls.h>
#include <asm/syscall_wrapper.h>
#include <linux/sched/mm.h>

#define __NR_add 549 //The number of system call table

void **syscall_table;
void *real_add;

SYSCALL_DEFINEx(1, _os_info, pid_t, pid)
{
        struct vm_area_struct *list;
        char str[100]={'\0'};
        char *path_name = NULL;

        /////  Referencing the class materials  /////
        struct task_struct *t;
        struct mm_struct *mm;
        t = pid_task(find_vpid(pid), PIDTYPE_PID);
        mm = get_task_mm(t);
        ////////////////////////////////////////////

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

        list=mm->mmap; //start point

        while(list) //For check all list
        {
                if(list->vm_file != NULL) //For checking exist file
                {
                        path_name = dentry_path_raw(list->vm_file->f_path.dentry, str, sizeof(str)); //path
                        printk(KERN_INFO "mem[%lx~%lx] code[%lx~%lx] data[%lx~%lx] heap[%lx~%lx] %s"
                                ,list->vm_start, list->vm_end, mm->start_code
                                ,mm->end_code, mm->start_data, mm->end_data, mm->start_brk, mm->brk, path_name); //Print information of process
                }
                list = list->vm_next; //move pointer
        }
        printk(KERN_INFO "#############################################################\n");

        mmput(mm);
        return 0;
}
```

위 코드는 wrapping할 시 add함수가 동작하게 되는 code이다. 과제 요구사항에 따라 **빨간 네모**는 process ID를 인자로 받으며, 0을 반환하는 동작을 표시한 것이며, **노란색 네모**는 정보 출력에 있어 printk함수를 사용하며, KERN_INFO를 함께 사용한 것을 확인할 수 있다. 해당 system call에서 쓰인 구조체와 함수는 아래와 같이 정리할 수 있다.

```c
/////  Referencing the class materials  /////
struct task_struct *t;
struct mm_struct *mm;
t = pid_task(find_vpid(pid), PIDTYPE_PID);
mm = get_task_mm(t);
////////////////////////////////////////////
```

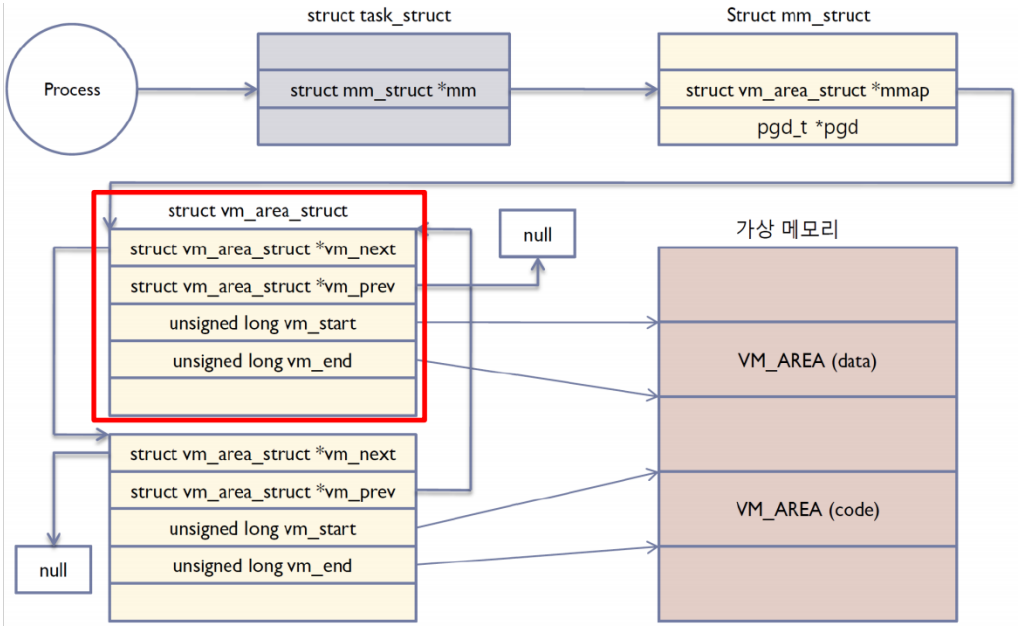
위 code는 참고자료에서 제공받은 함수들이며, 강의 자료를 기반하면 **task_struct** 구조체를 선언해줌으로써 구조체를 사용할 준비를 하고 **fin_vpid()**함수를 통하여 PID에 해당 하는 구조체를 반환하며, 그에 기반하여 **pid_task()**함수를 사용하여, 그에 관련한 첫 번째 task 구조체를 반환 받는다. 다음으로 **mm_struct** 구조체를 선언하고, **get_task_mm()** 함수를 사용하여, task의 **mm_struct**를 불러온다. 여기까지 code가 진행 되었다면, 강의 자료에서 제공받은 다음과 같은 그림까지 진행 된 상태이다.




```
struct vm_area_struct *list;
```

다음은 각 영역을 출력하기 위한 list정보를 담기 위한 구조체를 선언하며, 해당 함수에 mm_struct의 mmap를 저장하면서, list의 첫 번째 주소를 할당 할 수 있다.

다음 함수는 할당해주면서 그림은 다음과 같이 이해할 수 있다.



위 과정과 같이 정보가 위치하는 메모리 주소와 프로세스의 정보를 출력하기 위한 준비가 끝났다면 list가 NULL이 될 때까지 아래 code를 참고하여 정보를 출력하면 된다.

```
path_name = dentry_path_raw(list->vm_file->f_path.dentry, str, sizeof(str)); //path
```

위 코드는 file구조체를 사용하여 dentry정보를 기반으로 전체 경로를 반환 받는 함수 dentry_path_raw를 사용한 함수이다. 인자로는 현재 list의 dentry와 buffer, buffer의 size가 인자로 전달되며, path_name변수에 전체 경로가 반환되게 된다.

```
printk(KERN_INFO "mem[%lx~%lx] data[%lx~%lx] code[%lx~%lx] heap[%lx~%lx] %s"
        ,list->vm_start, list->vm_end, mm->start_data, mm->end_data, mm->start_code
        ,mm->end_code, mm->start_brk, mm->brk, path_name); //Print information of process
```

위 code는 각 list의 정보를 출력해주는 함수로써 출력 format은 16진수를 사용하였으며, 각각 정보에 대한 접근은 강의 자료에서 참고할 수 있으며, 간단한 설명은 다음과 같다.

Vm_start & vm_end → 영역의 시작주소, 끝 주소

Start_data & end_data → 데이터 세그먼트 영역

Start_code & end_code → 코드 세그먼트 영역

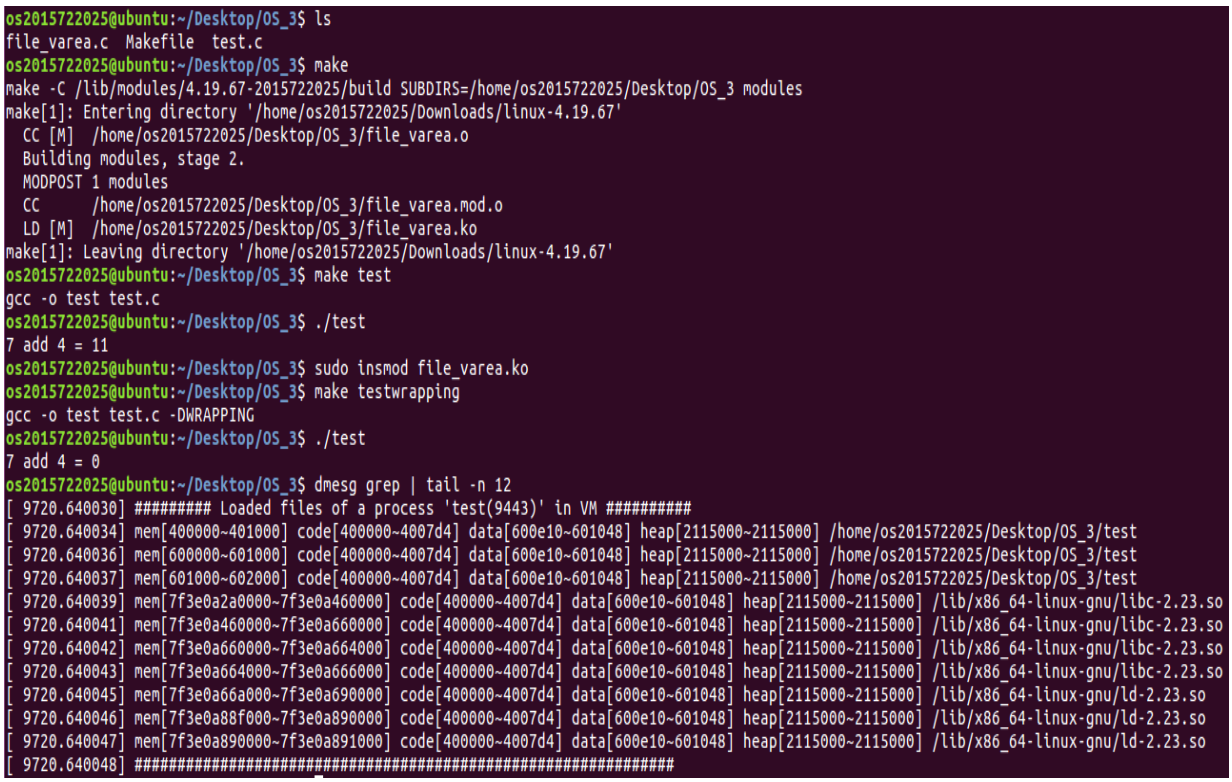
Start_brk & brk → heap의 시작과 끝 주소를 가지는 변수

```
void make_rw(void *addr) //For giving permission reading writting
{
        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) //recall read-write permissions
{
        unsigned int level;
        pte_t *pte = lookup_address((u64)addr, &level);

        pte->pte = pte->pte &~ _PAGE_RW;
}

static int __init hooking_init(void) //The function is called when loading a module
{
        syscall_table = (void**) kallsyms_lookup_name("sys_call_table");

        make_rw(syscall_table);

        real_add = syscall_table[__NR_add];
        syscall_table[__NR_add] = __x64_sys_os_info; //Insert function of __x64sys_os_info
        printk(KERN_INFO "init_[2015722025]\n");
        return 0;
}

static void __exit hooking_exit(void) //The function is called when clear module
{
        syscall_table[__NR_add] = real_add;

        make_ro(syscall_table);
        printk(KERN_INFO "exit_[2015722025]\n");
}

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

위 code는 이전 과제에서 사용하였던 code를 해당 과제를 진행하면서 다시 사용할 수 있도록 약간 수정하여 그대로 사용하였다.

(2) 모듈 적재 후 test 코드를 실행하여 시스템 콜을 실행시키면 dmesg로 정보를 확인

다음 그림은 제시된 과제의 실행 결과를 그대로 따라 출력한 것이다.



위 사진과 같이 정보 출력이 잘 나온 것을 확인할 수 있다.

3. Consideration

강의 자료를 참고하였기에 각각의 구조체를 연결하여 사용하는 것은 어렵지 않았습니다. 또한 참고하여 연결한 리스트에 대해서도 기존의 linked list 개념을 활용하여 다음 page에 접근하는 방식을 익히고 있었기에 어렵지 않았으며, 출력되는 정보 또한 함수 실행 위치에 따라 주소가 변동되는 모습을 확인했으며, 이는 이론 수업과 관련하여 directory위치에 따라 할당하는 memory위치가 다르다는 것을 알게 되었습니다. 반면 전체 경로를 반환 받기 위한 함수를 찾는데 어려움이 있었습니다. Reference에 올려놓은 site를 통하여 library함수를 검색하여 사용하는 구조체가 포함하고 있는 구조체(file 구조체)를 확인할 수 있었으며, 또한 file 구조체가 dentry정보를 포함하고 있는 것과 dentry를 이용하면 경로를 도출할 수 있는 함수를 dentry_path_raw함수를 linux관련 Q&A페이지에서 확인할 수 있었습니다. 이런 과정에서 시간이 조금 소요되었으며, 나머지 과정은 수월한 과제였습니다.

4. Reference

[1] 제공된 강의 자료

[2] linux관련 library정보 (vm_file에 대한 정보를 찾음)

https://elixir.bootlin.com/linux/v4.19.67/source/include/linux/mm_types.h

[3] 전체 경로를 찾기 위한 참고 site (file 구조체, dentry 관련)

1. https://stackoverflow.com/questions/33249643/how-to-use-dentry-path-raw

2. https://www.linuxquestions.org/questions/linux-kernel-70/how-to-find-the-complete-file-path-using-struct-file-in-linux-kernel-modules-4175501975/

3. https://sonseungha.tistory.com/123 (file 구조체)