

Лабораторная работа №2

По операционным системам

Вариант «syscall: memblock, vm_area_struct»

Студент: Ингликова С.

Группа: Р33312

Преподаватель: Осипов С.В.

г. Санкт-Петербург

2022 г

Задание

Разработать комплекс программ на пользовательском уровне и уровне ядра, который собирает информацию на стороне ядра и передает информацию на уровень пользователя, и выводит ее в удобном для чтения человеком виде. Программа на уровне пользователя получает на вход аргумент(ы) командной строки (не адрес!), позволяющие идентифицировать из системных таблиц необходимый путь до целевой структуры, осуществляет передачу на уровень ядра, получает информацию из данной структуры и распечатывает структуру в стандартный вывод. Загружаемый модуль ядра принимает запрос через указанный в задании интерфейс, определяет путь до целевой структуры по переданному запросу и возвращает результат на уровень пользователя.

Выполнение

1. memblock

Чтобы структура не затиралась после загрузки операционной системы, следует, чтобы в файле .config был указан параметр:

```
CONFIG_ARCH_KEEP_MEMBLOCK=y
```

Его дефолтное значение (которое используется при создании конфигурации) хранится в файле <папка с кодом ядра>/mm/Kconfig.

```
Symbol: ARCH_KEEP_MEMBLOCK [=n]
Type   : bool
Defined at mm/Kconfig:139
```

Надо поставить там значение «у» для ARCH_KEEP_MEMBLOCK.

Кроме того, чтобы структура была определена при обращении к ней, надо добавить экспорт символа в файл mm/memblock.c

Код системного вызова:

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

SYSCALL_DEFINE0(memblock)
{
    printk(KERN_INFO "memblock \n");
    printk(KERN_INFO "bottom up: %s \n", ((int)memblock.bottom_up==
0)?"no":"yes");
    printk(KERN_INFO "current limit: 0x%llx \n",
(u64)memblock.current_limit);
    printk(KERN_INFO "memblock types: \n");
    printk(KERN_INFO "1.\n");
    printk(KERN_INFO "memory name: %s \n", memblock.memory.name);
    printk(KERN_INFO "total_sz: 0x%llx \n",
(u64)memblock.memory.total_size);
    printk(KERN_INFO "count: %lu\n", memblock.memory.cnt);
    printk(KERN_INFO "2.\n");
    printk(KERN_INFO "memory name: %s \n", memblock.reserved.name);
    printk(KERN_INFO "total_sz: 0x%llx \n",
(u64)memblock.reserved.total_size);
    printk(KERN_INFO "count: %lu \n", memblock.reserved.cnt);
    return 0;
}
```

2. vm_area_struct

С этой структурой никаких подводных камней нет. В моей реализации выводится первая `vm_area_struct` для процесса с указанным пользователем `pid` (если такая есть).

Надо отметить, что сама структура содержит много полей, и я вывожу лишь основные из них.

Код системного вызова:

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

SYSCALL_DEFINE1(vm_area_struct, int, pid_val)
{
    struct pid* pid_val;
    struct task_struct* process;
    struct mm_struct* mm;
    struct vm_area_struct * vm_area;

    pid_val = find_get_pid(1);
    process = get_pid_task(pid_val, PIDTYPE_PID);
    if (process == NULL) {
        printk(KERN_INFO "no task with gived pid \n");
        return 0;
    }
    mm = process->mm;
    if (mm == NULL) {
        printk(KERN_INFO "no memory mappings \n");
        return 0;
    }
    vm_area = mm->mmap;

    printk(KERN_INFO "VM area struct \n");
    printk(KERN_INFO "vm area start: 0x%lx \n", vm_area->vm_start);
    printk(KERN_INFO "vm area end: 0x%lx \n", vm_area->vm_end);
    printk(KERN_INFO "vm area flags: 0x%lx\n", vm_area->vm_flags);
    printk(KERN_INFO "some basic characteristics: \n");
    if ((vm_area->vm_flags & 1) == 1) {
        printk(KERN_INFO "+ VM_READ\n");
    } else {
        printk(KERN_INFO "- VM_READ\n");
    }
    if ((vm_area->vm_flags & 2) == 2) {
        printk(KERN_INFO "+ VM_WRITE\n");
    } else {
        printk(KERN_INFO "- VM_WRITE\n");
    }
    if ((vm_area->vm_flags & 4) == 4) {
        printk(KERN_INFO "+ VM_EXEC\n");
    } else {
        printk(KERN_INFO "- VM_EXEC\n");
    }
    if ((vm_area->vm_flags & 8) == 8) {
        printk(KERN_INFO "+ VM_SHARED\n");
    } else {
        printk(KERN_INFO "- VM_SHARED\n");
    }
    printk(KERN_INFO "file-backed: %s \n", (vm_area->vm_file == NULL)?
    "no": "yes");
}
```

```

if (vm_area->vm_file != NULL) {
    printk(KERN_INFO "file name: %s \n", vm_area->vm_file-
>f_path.dentry->d_name.name);
    printk(KERN_INFO "file offset (in page sz units): 0x%lx \n",
vm_area->vm_pgoff);
}
return 0;
}

```

3. структура проекта

Для каждого системного вызова в папке с кодом ядра создается отдельная директория.

В ней должен лежать Makefile и файл с кодом системного вызова.

- linux_<version>
 - Прочие файлы
 - memblock
 - memblock.c
 - Makefile
 - vm_area_struct
 - vm_area_struct.c
 - Makefile

Каждый из созданных мейкфайлов содержит лишь одну строку:

obj-y := <название файла с кодом>.o

Также в основной Makefile ядра надо добавить названия созданных директорий:

*core-y += kernel/ certs/ mm/ fs/ ipc/ security/ crypto/ block/ **memblock/ vm_area_struct/***

4. добавление новых вызовов в системные таблицы

Перед компиляцией ядра надо совершить два шага:

- добавить интерфейсы системных вызовов в include/linux/syscalls.h:

```
asmlinkage long sys_memblock(void);
```

```
asmlinkage long sys_vm_area_struct(int);
```

- добавить новые системные вызовы в таблицу arch/x86/entry/syscalls/syscall_64.tbl:

```
436 common memblock      __x64_sys_memblock
```

```
437 common vm_area_struct __x64_sys_vm_area_struct
```

5. компиляция ядра

После внесения всех необходимых правок ядро конфигурируется и компилируется следующими командами:

```
sudo make defconfig
```

```
sudo make -j 8 (у меня виртуальной машине выданы 4 ядра, то есть 8 логических процессоров)
```

```
sudo make modules_install install
```

```
reboot (или ручная перезагрузка)
```

6. тестовая программа (на стороне пользователя)

Исходный код (usr_side.c):

```
#include <sys/syscall.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

void error_msg() {
    printf("Invalid prog usage. Possible variants:\n");
    printf("./usr_side -m\n");
    printf("./usr_side -vma <pid>\n");
}

int main(int argc, char *argv[])
{
    if (argc < 2) {
        error_msg();
    } else {

        int ans = -1;

        if (strcmp(argv[1], "-m") == 0) {
            ans = syscall(436);
        }

        if (strcmp(argv[1], "-vma") == 0) {
            if (argc == 3) {

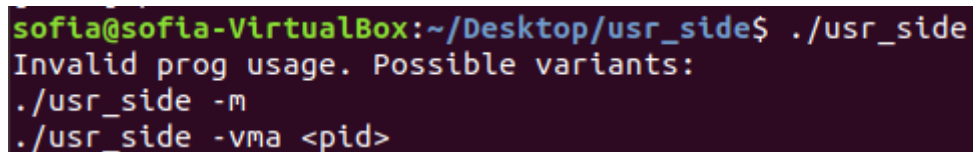
                int pid = atoi(argv[2]);
                ans = syscall(437, pid);

            } else {
                error_msg();
            }
        }

        if (ans == 0) {
            printf("SUCCESS\n");
        } else {printf("ERROR\n");}

    }
    return 0;
}
```

После компиляции и запуска с определенными параметрами нужно взглянуть в dmesg. Именно туда будет выведена информация о запрошенных структурах (для наглядности лучше предварительно очистить журнал с помощью sudo dmesg -C).



```
sofia@sofia-VirtualBox:~/Desktop/usr_side$ ./usr_side
Invalid prog usage. Possible variants:
./usr_side -m
./usr_side -vma <pid>
```

```
sofia@sofia-VirtualBox:~/Desktop/usr_side$ ./usr_side -m  
SUCCESS
```

```
sofia@sofia-VirtualBox:~/Desktop/usr_side$ sudo dmesg
```

```
[ 191.973328] memblock  
[ 191.973329] bottom up: no  
[ 191.973330] current limit: 0x120000000  
[ 191.973330] memblock types:  
[ 191.973330] 1.  
[ 191.973331] memory name: memory  
[ 191.973331] total_sz: 0xffff8ec00  
[ 191.973384] count: 3  
[ 191.973459] 2.  
[ 191.973460] memory name: reserved  
[ 191.973460] total_sz: 0xafdbbd0  
[ 191.973461] count: 40
```

```
sofia@sofia-VirtualBox:~/Desktop/usr_side$ ps aux | grep bash
```

```
sofia      2195  0.0  0.1  23748  5132 pts/0    Ss   16:27   0:00 bash  
sofia      2376  0.0  0.0  15648  1008 pts/0    S+   16:30   0:00 grep --auto bash
```

```
sofia@sofia-VirtualBox:~/Desktop/usr_side$ sudo dmesg -C
```

```
sofia@sofia-VirtualBox:~/Desktop/usr_side$ ./usr_side -vma 2195  
SUCCESS
```

```
sofia@sofia-VirtualBox:~/Desktop/usr_side$ sudo dmesg
```

```
[ 403.701371] VM area struct  
[ 403.701372] vm area start: 0x55915dda4000  
[ 403.701372] vm area end: 0x55915dea8000  
[ 403.701372] vm area flags: 0x875  
[ 403.701372] some basic characteristics:  
[ 403.701373] + VM_READ  
[ 403.701373] - VM_WRITE  
[ 403.701373] + VM_EXEC  
[ 403.701373] - VM_SHARED  
[ 403.701374] file-backed: yes  
[ 403.701375] file name: bash  
[ 403.701375] file offset (in page sz units): 0x0
```

Выводы

Во время выполнения лабораторной работы я написала системные вызовы, добавила их в ядро и перекомпилировала его, таким образом изучив один из способов взаимодействия стороны пользователя с ядром.