Assignment #1 - System Calls - REPORT

Name: Nguyen Le Huy - ID: 1611290

March 20, 2018

1 Adding a new System Call

1.1 Preparation

Setup Virtual Machine: You can use Virtual Box or VMware.
You can choose a version of Ubuntu image at https://www.osboxes.org/ubuntu/
Install packages: You have to install nessesary packages to compile and install kernel.

```
$ sudo apt-get update
$ sudo apt-get install build-essential kernel-package
$ sudo apt-get install libncurses5-dev openssl libssl-dev
```

QUESTIONS: Why do we need to install kernel-package?
ANSWER: Because kernel-package helps automate the routine steps required to compile and install a custom kernel.
Download and unpack kernel source: Version 4.4.56 is recommended:

```
$ $ mkdir ~/kernelbuild && cd ~/kernelbuild
$ wget https://cdn.kernel.org/pub/linux/kernel/v4.x/linux-4.4.56.tar.xz
$ tar -xJf linux-4.4.56.tar.xz
```

QUESTION: Why do we have to use another kernel source from the server such as `http://www.kernel.org`, can we compile the local kernel on the running OS directly?
ANSWER: Because local kernel doesn't have source files (.c), so we can't compile it.

1.2 Configuration

Copy content of configuration file:

```
$ cp /boot/config-4.x.x-generic ~/kernelbuild/[kernel directory]/.config
```

Note: run `uname -r` to see 4.x.x-generic. [kernel directory] will be `linux-4.4.56`.
Important: You must rename your kernel version, run terminal at [kernel directory] with:

```
$ make nconfig // or make menuconfig
```

Go to `"General setup"`, access to `"(-ARCH) Local version - append to kernel release"`, then enter `".MSSV"` with MSSV is your student ID. Press `"F6"` to save and `"F9"` to quit.
If you can't use `"make nconfig"`, run:

```
$ nano .config
```

then add your MSSV to `CONFIG_LOCALVERSION=".MSSV"` and save the file.
Goto arch/x86/entry/syscalls, add these to both file in this directory:
In `syscall_32.tbl`:

```
[number]    i386    procmem         sys_procmem
```

QUESTION: What is the meaning of other parts, i.e. i386, procmem, and `sys_procmem`?
ANSWER: `i386` is the application binary interface refers to the compiled binaries in machine code, `procmem` is the system call's name, `sys_procmem` is the entry point where system call is executed.
In `syscall_64.tbl`:

```
[number]    x32     procmem          sys_procmem
```

Open the file `include/linux/syscalls.h` and add the following lines to the end of this file:

```
struct proc_segs;
asmlinkage long sys_procmem(int pid, struct proc_segs *info);
```

Our system call will be implemented in `arch/x86/kernel` directory in a source file `sys_procmem.c`.
QUESTION: What is the meaning of each line above?
ANSWER: Define `proc_segs` struct and `sys_procmem` system call entry point, `sys_procmem` takes process ID and the address of variable type `struct proc_segs`, finds memory information of that process and store them to that variable, `asmlinkage` tells the compiler to look on the CPU stack for the function parameters, instead of registers.

2 System Call Implementation

2.1 Test System Call using modules

Before write down the system call in `sys_procmem.c`, you should test it first using kernel modules of the current kernel in your virtual machine.
Let's create a file called "`test.c`":

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/sched.h>
int init_module(void) {
    /*Put your code here to test*/
}
void cleanup_module(void) {
    printk(KERN_INFO "Bye.\n");
}
```

Create a Makefile:

```
obj-m   += test.o
all:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
clean:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

Then run these lines in terminal in directory contains those files:

```
$ make
$ sudo insmod test.ko
$ sudo rmmod test
$ tail /var/log/syslog
```

The 4th line will show you the result of function `init_module` and `cleanup_module`.

2.2 Implementation

To implement this system call, use `task_struct` and `mm_struct` defined in `include/linux/sched.h`:

```
struct task_struct {
    struct mm_struct *mm, *active_mm;
    pid_t pid;
    ...
};
struct mm_struct {
```

```
    unsigned long start_code, end_code, start_data, end_data;
    unsigned long start_brk, brk, start_stack;
    ...
};
```

Add these to file arch/x86/kernel/sys_procmem.c:

```
#include <linux/linkage.h>
#include <linux/sched.h>
struct proc_segs {
    unsigned long mssv; /*Your student ID*/
    unsigned long start_code, end_code, start_data, end_data;
    unsigned long start_heap, end_heap, start_stack;
};
asmlinkage long sys_procmem(int pid, struct proc_segs *info) {
    // TODO: Implement the system call
}
```

In the same folder of sys_procmem.c, add this line to the end of Makefile:

```
obj-y       += sys_procmem.o
```

3 Compilation and Installation Process

3.1 Compilation

In [kernel directory], run:

```
$ make -j 4
$ make -j 4 modules
```

QUESTION: What is the meaning of these two stages, namely "make" and "make modules"?
ANSWER: "make modules" just compiles the modules. "make" compiles the kernel and create vmlinuz (kernel image which will be uncompressed and loaded into memory by GRUB or any boot loader you use).

3.2 Installation

In [kernel directory], run:

```
$ sudo make -j 4 modules_install
$ sudo make -j 4 install
$ sudo reboot
```

Hold "shift" key while booting, choose "Advanced..." then choose kernel.
Check kernel version using uname -r.

3.3 Testing

QUESTION: Why this program could indicate whether our system call works or not?
ANSWER: Because if it works, it will print your student ID.

4 Making API for System Call

Create wrapper directory:

```
$ mkdir ~/wrapper
```

Put this code in procmem.h in wrapper:

```
#ifndef _PROC_MEM_H_
#define _PROC_MEM_H_
#include <unistd.h>
#include <sys/types.h>
struct proc_segs {
    unsigned long mssv; /*Your student ID*/
    unsigned long start_code, end_code, start_data, end_data;
    unsigned long start_heap, end_heap, start_stack;
};
long procmem(pid_t pid, struct proc_segs *info);
#endif // _PROC_MEM_H_
```

QUESTION: Why we have to re-define proc_segs struct while we have already defined it inside the kernel?
ANSWER: Because we can't get that struct in kernel directly, the compiler won't know where to find that struct if we don't re-define it.
Put this code in procmem.c in wrapper:

```
#include "procmem.h"
#include <linux/kernel.h>
#include <sys/syscall.h>
long procmem(pid_t pid, struct proc_segs *info) {
    // TODO: implement the wrapper here.
}
```

Copy our header file to header directory of our system and compile as a shared object:

```
$ cd ~/wrapper
$ sudo cp procmem.h /usr/include
$ gcc -shared -fpic procmem.c -o libprocmem.so
```

If the compilation ends successfully:

```
$ sudo cp libprocmem.so /usr/lib
```

QUESTION: Why root privilege (e.g. adding sudo before the cp command) is required to copy the header file to /usr/include?
ANSWER: Because only root user can write files and folders in /usr
QUESTION: Why we must put -share and -fpic option into gcc command?
ANSWER: Because -shared creates a shared library which is loaded by programs when they start. Using -fpic option usually generates smaller and faster code, but will have platform-dependent limitations, such as the number of globally visible symbols or the size of the code.