HCMC University Of Technology
Faculty of Computer Science & Engineering

Course: Operating Systems
Assignment #1 - System Call

Phuong-Duy Nguyen, Duc-Hai Nguyen, Minh Thanh Chung

September 17, 2016

Goal: This assignment helps students to understand steps of modifying, compiling and installing Linux kernel.

Content: In detail, student will add a new system call which helps applications to know the data layout (e.g. boundary of segments such as text, heap, etc.) of a given process. This task requires student to understand system call invocation mechanism as well as steps of compiling and installing Linux kernel.

Result: After this assignment, student should know how to modify Linux kernel and deploy their own kernel on a given machine.

Contents

1 Introduction

1.1 System calls

System calls provide an interface to the services made available by an operating system. These calls are generally available as routines written in C and C++, although certain low-level tasks (for example, tasks where hardware must be accessed directly) may have to be written using assembly-language instructions.
As you can see, even simple programs may make heavy use of the operating system. Frequently, systems execute thousands of system calls per second. Most programmers never see this level of detail, however. Typically, application developers design programs according to an application programming interface (API). The API specifies a set of functions that are available to an application programmer, including the parameters that are passed to each function and the return values the programmer can expect. The functions that make up an API typically invoke the actual system calls on behalf of the application programmer.

EXAMPLE OF STANDARD API As an example of a standard API, consider the `open()` function that is available in UNIX and Linux systems. The API for this function is obtained from the man page. A description of this API appears below:

```
$ man open

#include <unistd.h>

int open(const char *path, int oflags);
int open(const char *path, int oflags, mode_t mode);
```

A program that uses the `open()` function must include the unistd.h header file, as this file defines `int` data types (among other things). The parameters passed to `open()` are as follows.

- `const *path` - The relative or absolute path to the file that is to be opened.
- `int oflags` - A bitwise 'or' separated list of values that determine the method in which the file is to be opened.
- `mode_t mode` - A bitwise 'or' separated list of values that determine the permissions of the file if it is created.

The file descriptor returned is always the smallest integer greater than zero that is still available. If a negative value is returned, then there was an error opening the file.

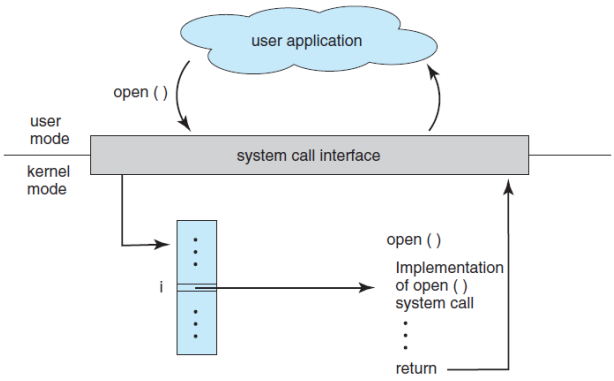

Figure 1: The handling of a user application invoking the open() system call.

The relationship between an API, the system-call interface, and the operating system is shown in Figure 1, which illustrates how the operating system handles a user application invoking the `open()` system call.

1.2 Requirement and Marking

As Figure 2 shown, this is the diagram of doing the assignment. Student will practice the progress of compiling Linux kernel. After that, the most important part is to implement a system call inside the kernel. The assignment is divided into multiple stages and score is marked by each stage as Figure 2.

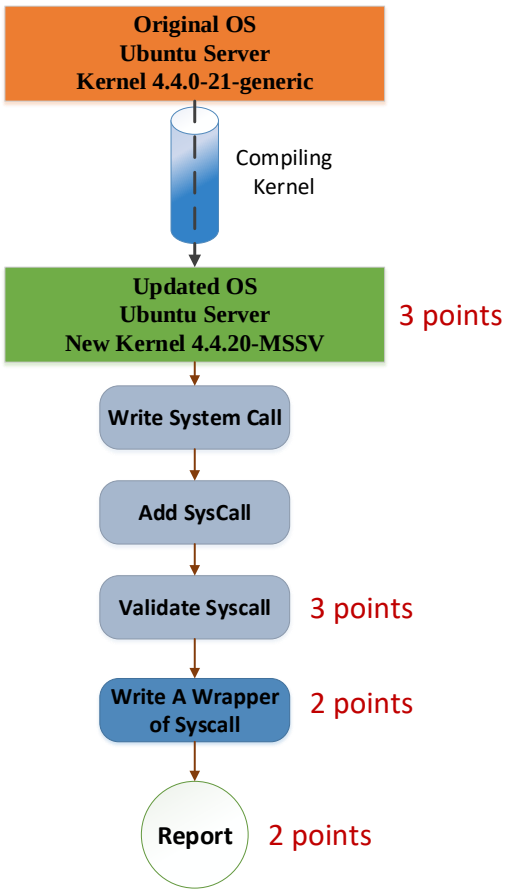

Figure 2: Diagram of implementing the assignment.

2 Compiling Linux Kernel

Compiling custom kernel has its own advantages and disadvantages. However, new Linux user/admin find it difficult to compile Linux kernel. Compiling kernel needs to understand few things and then just type couple of commands. This section guides you basic steps to compile the Linux kernel, but you need to consider the purposes of these commands for summarizing a short report.

2.1 Preparation

Set up Virtual machine Compiling and installing a new kernel is a risky task so you should work with the kernel inside a virtual machine. We have prepared an image file for a Ubuntu virtual machine running Linux kernel version 4.4.21. You must use this virtual machine in this assignment to ensure fairness. Besides, this virtual machine is set up for this assignment so we could minimize the risk of getting errors. You could get this virtual machine through the following link:
[LINK TO VIRTUAL MACHINE]
After downloading the file from this link to your own computer, you could use VMWare or Virtual Box to open this file. Originally, this virtual machine is equipped with 4 processors and 20GB disk space. This setting is sufficient for this assignment but it is fine if you change its hardware setting to adapt to your own computer.
After booting the virtual machine, log in to the system using the following information:

```
username: student
password: 123456
```

User student is a sudoer so you can run commands on the behalf of user root by adding "sudo" before the command.
Important: Because making a mistake when compiling or installing a new kernel could cause the entire machine to crash, you must strictly follow instructions in this section. We also encourage you to frequently take snapshots to avoid repeating time consuming tasks and quickly restore the virtual machine.

Install the core packages Get Ubuntu's toolchain (gcc, make, and so forth) by installing the build-essential metapackage:

```
$ sudo apt-get update
$ sudo apt-get install build-essential
```

Install kernel-package:

```
$ sudo apt-get install kernel-package
```

QUESTION: Why we need to install kernel-package?

Create a kernel compilation directory: It is recommended to create a separate build directory for your kernel(s). In this example, the directory kernelbuild will be created in the home directory:

```
$ mkdir ~/kernelbuild
```

Download the kernel source: *Warning:* systemd requires kernel version 3.11 and above (4.2 and above for unified *cgroups* hierarchy support). See `/usr/share/systemd/README` for more information. In this assignment, you should choose the version 4.4.21 for consistency.
Download the kernel source from http://www.kernel.org. This should be the tarball (tar.xz) file for your chosen kernel. It can be downloaded by simply right-clicking the tar.xz link in your browser and selecting

Save Link As..., or any other number of ways via alternative graphical or command-line tools that utilize HTTP, FTP, RSYNC, or Git.
In the following command-line example, wget has been installed and is used inside the `~/kernelbuild` directory to obtain kernel 4.4.21.

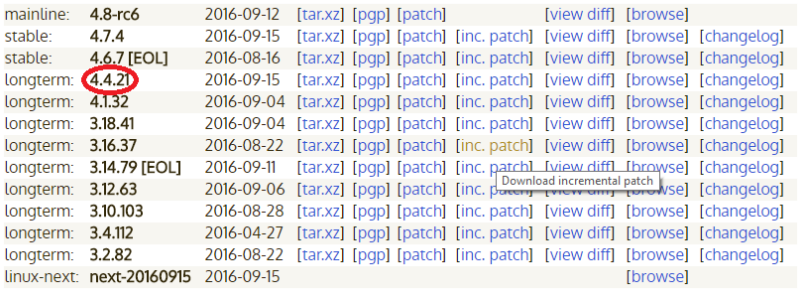

Figure 3: Kernel sources from www.kernel.org.

```
$ cd ~/kernelbuild
$ wget https://cdn.kernel.org/pub/linux/kernel/v4.x/linux-4.4.21.tar.xz
```

QUESTION: Why we have to use another kernel source from the server such as http://www.kernel.org, can we compile the original kernel (the local kernel on the running OS) directly?

Unpack the kernel source:
Within the build directory, unpack the kernel tarball:

```
$ tar -xvJf linux-4.4.21.tar.xz
```

2.2 Configuration

This is the most crucial step in customizing the default kernel to reflect your computer's precise specifications. Kernel configuration is set in its .config file, which includes the use of Kernel modules. By setting the options in .config properly, your kernel and computer will function most efficiently. Since making our own configuration file is a complicated process, we could borrow the content of configuration file of an existing kernel currently used by the virtual machine. This file is typically located in `/boot/` so our job is simply copy it to the source code directory:

```
$ cp /boot/config-4.4.0-21-generic ~/kernelbuild/.config
```

Note: 4.4.0-21-generic is the version of the kernel installed in the virtual machine. If you use other machine, please check it out by running `uname -r`.
Important: Do not forget to rename your kernel version in the General Setup. Because we reuse the configure file of current kernel, if you skip this, there is the risk of overwriting one of your existing kernels by mistake. To edit configure file through terminal interface, we must install *libncurses5 – dev* package first:

```
$ sudo apt-get install libncurses5-dev
```

Then, run `$ make menuconfig` or `$ make nconfig` to open Kernel Configuration.

```
$ make nconfig
```

To change kernel version, go to General setup option, Access to the line "(-ARCH) Local version - append to kernel release". Then enter a dot "." followed by your MSSV. For example:

```
.1401234
```

Press F6 to save your change and then press F9 to exit.
Note: During compiling, you can encounter the error caused by missing `openssl` packages. You need to install these packages by running the following command:

```
$ sudo apt-get install openssl libssl-dev
```

2.3 Build the configured kernel

First run "`make`" to compile the kernel and create `vmlinuz`. It takes a long time to "`$ make`", we can run this stage in parallel by using tag "`-j np`", where `np` is the number of processes you run this command.

```
$ make
or
$ make -j 4
```

`vmlinuz` is "the kernel". Specifically, it is the kernel image that will be uncompressed and loaded into memory by GRUB or whatever other boot loader you use.
Then build the loadable kernel modules. Similarly, you can run this command in parallel.

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

QUESTION: What is the meaning of these two stages, namely "`make`" and "`make modules`"?

2.4 Installing the new kernel

First install the modules:

```
$ sudo make modules_install
or
$ sudo make -j 4 modules_install
```

Then install the kernel itself:

```
$ sudo make install
or
$ sudo make -j 4 install
```

Check out your work: After installing the new kernel by steps described above. Reboot the virtual machine by typing

```
sudo reboot
```

After logging into the computer again, run the following command.

```
uname -r
```

If the output string contains your MSSV then it means your custom kernel has been compiled and installed successfully. You could progress to the next part.

3 System call

3.1 The role of system call

The main part of this assignment is to implement a new system call that lets the user to know about the memory organization of a process that is currently running on the system. The information about the process's memory layout is represented through the following struct:

```
struct prog_segs {
      unsigned long mssv;
      unsigned long start_code;
      unsigned long end_code;
      unsigned long start_data;
      unsigned long end_data;
      unsigned long start_heap;
      unsigned long end_heap;
      unsigned long start_stack;
};
```

`start_code` and `end_code` are two pointers that point to the first and the last byte of code segment, respectively. `start_data` and `end_data` point to the first and the last byte of data segment. Similarly, `start_heap` and `end_heap` point out the boundary of heap region. Finally, `start_stack` point to the first byte of stack segment. `mssv` is an additional field that is only used for marking purpose.

3.2 Prototype

The prototype of our system call is described as below:

```
long procmem(int pid, struct pro_segs * info);
```

To invoke `procmem` system call, user must provide the PID of the process from which it wants to get information through "`pid`" parameter. If the `procmem` system call finds out the process having given PID, it will get memory layout information of this process, put it in output parameter "info" and return 0. However, if the system call cannot find such process, it will return -1.

3.3 Implementation

Before implementing this system call, we must add it to the kernel first. Modern processors support invoking system calls in many different ways depend on their architecture. Since our virtual machine runs on x86 processors, we only consider about Linux's system call implementation for this architecture.

The list of system calls implemented for x86 architecture is located in `arch/x86/entry/syscalls`. For historical reason, Linux has different system call lists for 32-bit x86 processors and 64-bit x86 processors. Thus, in this directory, we have two lists in two separated files: `syscall_32.tbl` and `syscall_64.tbl`. To ensure our system call work well in every x86 processors, we must add it to both file.

In those file, each system call is declared in one row with following information: number, ABI, name, entry point and compat entry point separated by a TAB. System calls are call from user space through their numbers so our system call's number must be unique. To add our new system call, add the following line to the end of `syscall_32.tbl`:

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

Number is a value depend on the kernel version you are currently working on. However, choosing the number that equal to the largest number in the list plus one would be fine.
QUESTION: What is the meaning of other parts, i.e. i386, procmem, and `sys_procmem`?

Similarly, add the following line to the end of syscall_64.tbl:

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

At this time, we have told the kernel that we have a new system call to be deployed but we still do not let it know the definition (e.g. its input parameters, return values, etc.) of this new system call. The next job is to explicitly define this system call. To do so, we must add necessary information to kernel's header files. Open the file `include/linux/syscalls.h` and add the following line to the end of this file:

```
struct proc_segs;

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

QUESTION: What is the meaning of each line above?

We now implement our system call. Inside kernel directory, create a new source file named `sys_procmem.c`. In this file, add the following lines.

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

In the body part of the function `sys_procmem` is your code that use to realize our system call. **Remember** to put your MSSV to *mssv* field of output parameter "`info`".
After finishing your job, recompile and re-install kernel by repeating steps in Section 2. After booting to the new kernel, you could create a small C program to check your work.

```
#include <sys/syscall.h>
#include <stdio.h>
#define SIZE   10

int main() {
     long sysvalue;
     unsigned long info[SIZE];
     sysvalue = syscall([number_32], 1, info);
     printf("My MSSV: %ul\n", info[0]);
}
```

Remember to replacing [`Number_32`] by the number of `procmem` system call in the file `syscall_32.tlb`
QUESTION: Why this program could indicate whether our system works or not?

Although `procmem` system call works properly, we still have to invoke it through its number which is quite inconvenient for other programmers so we need to implement a C wrapper for it to make it easy to use. This can be done outside the kernel. Thus, to avoid recompile the kernel again, we leave out kernel source code directory and create another directory to store source code for our wrapper. We first create a header file which contains the prototype of the wrapper and declare `proc_segs` struct. Naming the it with `procmem.h` and put the following lines into its content:

```
#ifndef _PROC_MEM_H_
#define _PROC_MEM_H_
#include <unistd.h>

struct proc_segs {
    unsigned long mssv;
    unsigned long start_code;
    unsigned long end_code;
    unsigned long start_data;
    unsigned long end_data;
    unsigned long start_heap;
    unsigned long end_heap;
    unsigned long start_stack;
};
long procmem(pid_t pid, struct prog_segs * info);
#endif // _PROC_MEM_H_
```

Note: You must define fields in `proc_segs` struct in the same order as you did in the kernel.
QUESTION: Why we have to re-define `proc_segs` struct while we have already defined it inside the kernel?

We then create a file named `procmem.c` to hold the source code file for wrapper. The content of this file should be as follows:

```
#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.
}
```

Hint: You could implement your wrapper based on the code of our test program above.

3.4 Validation

You could check your work by write an additional test module to call this functions but do not include the test part to your source file (`procmem.c`). After making sure that the wrapper work properly, we then install it to our virtual machine. First, we must ensure everyone could access this function by making the header file visible to GCC. Run following command to copy our header file to header directory of our system:

```
$ sudo cp <path to procmem.h> /usr/include
```

QUESTION: Why root privilege (e.g. adding sudo before the cp command) is required to copy the header file to `/usr/include`?
We then compile our source code as a shared object to allow user to integrate our system call to their applications. To do so, run the following command:

```
$ gcc -share -fpic procmem.c -o libprocmem.so
```

If the compilation ends successfully, copy the output file to `/usr/lib`. (Remember to add `sudo` before `cp` command).
QUESTION: Why we must put `-share` and `-fpic` option into `gcc` command?.

We only have the last step: check all of your work. To do so, write following program, and compile it with `-lprocmem` option. The result should be consistence with the content of `/proc/<pid>/maps` file.

```c
#include <procmem.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdint.h>

int main() {
     pid_t mypid = getpid();
     printf("PID: %d\n", mypid);
     struct prog_segs info;

     if (procmem(mypid, &info) == 0) {
          printf("Code segment: %lx-%lx\n", info.start_code, info.end_code);
          printf("Data segment: %lx-%lx\n", info.start_data, info.end_data);
          printf("Heap segment: %lx-%lx\n", info.start_heap, info.end_heap);
          printf("Start stack: %lx\n", info.start_stack);
     }else{
          printf("Cannot get information from the process %d\n", mypid);
     }
     // If necessary, uncomment the following line to make this program run
     // long enough so that we could check out its maps file
     // sleep(100);
}
```

4 Submission

4.1 Source code

After you finish the assignment, you need to compress the following code files into `assignment1_MSSV.zip`:

- sys_procmem.c
- procmem.c
- report.pdf (Your report in PDF format)

Requirement: you have to code the system call followed by the coding style. Reference: https://www.gnu.org/prep/standards/html_node/Writing-C.html.

4.2 Report

As Figure 2 shown, the score for compiling kernel is 2 points, System call is 3 points, Wrapper of System call is 2 points and the report is 3 points. For the content of assignment report, describe steps of adding procmem system call to Linux kernel. The report layout is follows: `assignment1_MSSV.zip`:

- Adding new system call
- System call Implementation
- Compilation and Installation process
- Making API for system call

In the report, just describe steps in short, succinct paragraphs. You have to add your answer to questions with the highlight word "**QUESTION**" throughout this instruction to related sections. Please do not answer questions as a list of items. Your score are given based on the correctness of your answers and the clarify of the report content.