HCMC University Of Technology
Faculty of Computer Science & Engineering

[CO2018] Operating System Lab – SPRING 2019

Assignment #1
– System Call –

May 1, 2019

Goal:

1. Understand the process of modifying, compiling and installing Linux kernels.
2. Understand system calls and how to develop them.

Content: Students will be required to add a new system call that provides information about the memory layout of a given process.

Outcome: Students are expected to possess solid technical skills about modifying Linux kernels and deploying their own kernels on any machine.

Due date: 22-May-2019

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:

Listing 1: Declaration of `open()`

```
$ 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 the integer that greater than or equal to zero then the file opening is success. If a negative value is returned, then there was an error occur while opening the file.

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

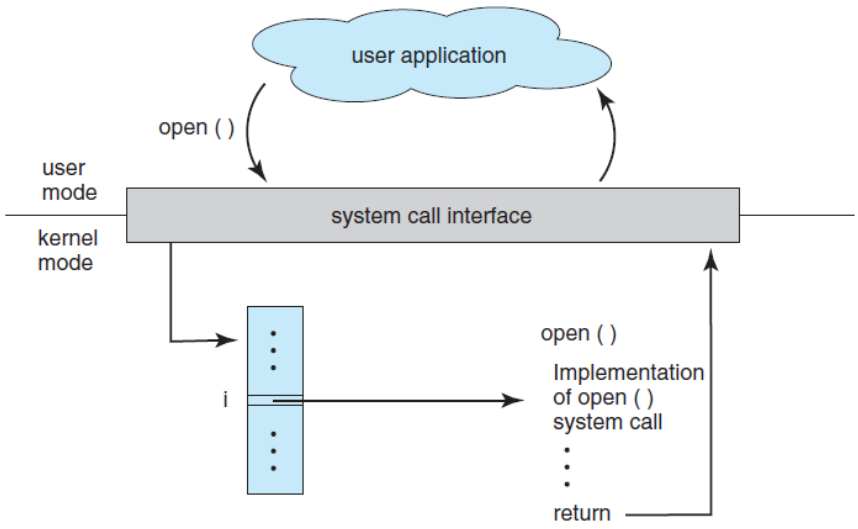

Figure 1: The handling of a user application invoking the open() system call.

1.2 Overview and Grading Policy

An overview of this assignment and grading scales of each stage are as follows:

1. Prepare a virtual machine
2. Develop the system call - **2 points**
3. Add the system call to the kernel
4. Compile a new kernel - **2 points**
5. Write a wrapper for the system call
6. Validate the system call - **2 points**
7. Write a report - **4 points**

The results of your work will be graded based on your Sakai submission and your virtual machine's new kernel (details on grading will be provided in the final section).

After this section, the rest of this document is organized in the following manners. Section 2 gives some information about preparing the virtual machine. The required system call is illustrated in Section 3. Section 4 provides steps towards compiling kernels. Section 5 guides you on how to write a wrapper for your system call. Section 6 helps you validate your work. Finally, Section 7 presents submission information.

2 Prepare Linux Kernel

2.1 Preparation

Set up Virtual machine: Compiling and installing a new kernel is a risky task so you should work with the kernel on a virtual machine (VM). We have prepared an image file for a VMWare Ubuntu VM, and we **STRONGLY recommend** you using this image which can be downloaded at BKeL. The user and password of this VM are **student**. User student is a sudoer so you can run commands on the behalf of user root by adding "sudo" before the command.

Otherwise, if you wish to use your own VM, so you can choose the version of Ubuntu image for your laptop at: http://www.osboxes.org/ubuntu/ You should choose the version 12.04, because it doesn't require a high level of hardware in you laptop.

Important: Since even a small mistake when compiling or installing a new kernel could cause the entire machine to crash, you must strictly follow guidelines in this assignment. Furthermore, we urge you to frequently take snapshots to avoid repeating time consuming tasks or to quickly restore the VM.

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 do 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 or above (4.2 or above for unified *cgroups* hierarchy support). See `/usr/share/systemd/README` for more information. In this assignment, you should choose the version **4.4.56** for consistency.

Download the kernel source from http://www.kernel.org. This should be a tarball (tar.xz) file for your chosen kernel.

In the following command-line example, **wget** has been used within the *~/kernelbuild* directory to obtain the kernel 4.4.56.

```
$ cd ~/kernelbuild
$ wget https://cdn.kernel.org/pub/linux/kernel/v4.x/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 just compile the original kernel (the local kernel on the running OS) directly?

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
```

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

```
$ tar -xvJf linux-4.4.56.tar.xz
```

Note: from now on, for simplicity, we will use the terms "top directory" and "top directory of kernel source code" interchangeably to refer to the directory created as a result of extracting this tarball.

2.2 Configuration

This is the most crucial step in customizing the default kernel. 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 perform efficiently.

Since making our own configuration file is a complicated process, we could reuse the content of the configuration file of an existing kernel currently used by the virtual machine. This file is typically located in `/boot/` so our job is to simply copy it to the source code directory:

```
$ cp /boot/config-x.x.x-x-generic ~/kernelbuild/linux-4.4.56/.config
```

Note: x.x.x-x-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 a risk of overwriting one of your existing kernels by mistake. To edit the configure file through the terminal interface, we must install $libncurses5 - dev$ package first:

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

Then, run `$ make menuconfig` or `$ make nconfig` inside the top directory to open Kernel Configuration.

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

To change the 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 Student ID. For example:

```
.1601234
```

Press F6 to save your change and then press F9 to exit.

The purpose of this step is to change the name of the kernel after you compile. If you cannot use `make nconfig`, you can directly change the name of the kernel:

```
$ vim .config // change the content of this file
// Add your Student ID into the line
CONFIG_LOCALVERSION=".ID"
// Then save the file
```

3 System Call - procmem

In this assignment, you have to define a System Call named **procmem**. This syscall helps users show the memory layout of a specific process. For example:

```
Code Segment start = 0x8048000, end = 0x809fc38
Data Segment start = 0x80a0000, end = 0x80a0ec4
Stack Segment start = 0xbffffb30
```

To implement this system call, you have to use 2 data structures defined by Linux OS, `task_struct` and `mm_struct`.

In a Linux kernel, every process is associated with a struct `task_struct`. The definition of this struct is in the header file <linux/sched.h>

Listing 2: task_struct

```
struct task_struct {
     volatile long state; /* -1 unrunnable, 0 runnable, >0 stopped */
     struct thread_info *thread_info;
     atomic_t usage;
     ...
     ...
     struct mm_struct *mm, *active_mm;
     ...
     pid_t pid;
     ...
     char comm[16];
     ...
};
```

The `mm` field in `task_struct` points to the **memory descriptor**, `mm_struct`, which is an executive summary of a program's memory. The `mm_struct` is defined in <linux/mm_types.h> as follows:

Listing 3: mm_struct

```
struct mm_struct {
     struct vm_area_struct *mmap; /* list of VMAs */
     struct rb_root mm_rb;
     struct vm_area_struct *mmap_cache; /* last find_vma result */
     ...
     ...
     ...
     /* Process memory layout */
     unsigned long start_code , end_code , start_data , end_data;
     unsigned long start_brk , brk , start_stack;
     ...
     ...
     ...
};
```

Note: You should investigate more information about those data structures to find out which variable correspond to the code, data or stack segment.

3.1 Kernel Module

Since adding your system call to the kernel is a time-consuming task (for each time you fix your code, it has to compile the whole kernel), **Linux Kernel Module** is a simple way to test the logic and flow of your code before writing the system call. Specifically, modules are pieces of code that can be loaded and unloaded into the kernel upon demand. They extend the functionality of the kernel without the need to reboot the system [1].

Note: You should test Kernel Module in other folder.

For example: Hello, World (The Simplest Module) - http://www.tldp.org/LDP/lkmpg/2.6/html/x121.html. With regard to compiling and running the module, refer to http://www.tldp.org/LDP/lkmpg/2.6/html/x181.html

A kernel module must have at least two functions: a "start" (initialization) function called `init_module()` which is invoked when the module is `insmod` into the kernel, and an "end" (cleanup) function called `cleanup_module()` which is invoked just before it is `rmmod`.

With this assignment, you can use Linux Kernel Module to test your code before writing the syscall and compiling it. Listing 4 presents a skeleton code for this task, `pid` is the input argument of this module (you can refer to [2] for how to pass arguments)

Listing 4: Linux kernel module to test syscall

```
#include <linux/module.h>    // included for all kernel modules
#include <linux/kernel.h>    // included for KERN_INFO
#include <linux/init.h>      // included for __init and __exit macros
#include <linux/mm.h>
#include <linux/sched.h>

static int pid = 1;

static int __init procmem_init(void)
{
     struct task_struct *task;

     printk(KERN_INFO "Starting kernel module!\n");
     // TODO: find task_struct that is associated with the input process pid
     // Hint: use the for_each_process() function
     ...
     // TODO: show its memory layout
     ...
     return 0;
}
```

[1] http://www.tldp.org/LDP/lkmpg/2.6/html/x40.html
[2] http://www.tldp.org/LDP/lkmpg/2.6/html/x323.html

```
static void __exit procmem_cleanup(void)
{
     printk(KERN_INFO "Cleaning up module.\n");
}

module_init(procmem_init);
module_exit(procmem_cleanup);
module_param(pid, int , 0);
```

Note: After you have tested the flow and logic of your code with the Linux kernel module, you can proceed to write the required system call.

3.2 System call prototype

The prototype of the system call is described as follows:

Listing 5: System call prototype

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

To invoke the `procmem` system call, users must provide the PID of the process from which they want to get information through the "`pid`" parameter. If the `procmem` have found the process associated with the given PID, it will get the memory layout information of that process, put it in the 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, you must go back to top directory and add the system call's information to the kernel. Modern processors support invoking system calls in many different ways depending on the architecture. Since our virtual machine runs on x86 processors, we only concern Linux's system call implementation for this architecture.

The list of system calls implemented for x86 architecture is located in `arch/x86/entry/syscalls`. 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 all x86 processors, we must add it to both files.

In those files, each system call is declared in one row with the following information: number, ABI, name, entry point and compat entry point separated by a tab. System calls are invoked from user space through their numbers so the 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 which depends on the kernel version you are currently working on. However, you could simply choose a number that is equivalent to the largest number in the list plus one.

QUESTION: What is the meaning of other components, i.e. i386, `procmem`, and `sys_procmem`?

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

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

At this time, we have informed the kernel that we have a new system call to be deployed but we still have not configured its definition (e.g. its input parameters, return values, etc.) . Therefore, the next task is to explicitly define this system call. In other words, we must add necessary information to kernel's header files. Open the file `include/linux/syscalls.h` and add the following lines to the end of that file:

```
struct proc_segs;

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

QUESTION: What is the meaning of each line above?

Now, we implement our system call. Go to `arch/x86/kernel`, create a new source file named `sys_procmem.c`. In this file, add the following lines.

Listing 6: sys_procmem.c

```
#include <linux/linkage.h>
#include <linux/sched.h>
struct proc_segs {
     unsigned long studentID;
     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 function `sys_procmem`, put your code that realizes the system call and fills the `info`. **REMEMBER** to place your Student ID to the field `studentID`.

Finally, we have to inform the compiler to include our new source file in the compilation process when we rebuild the kernel. In the folder `arch/x86/kernel`, you need to add a line at the end of Makefile:

```
obj-y     += sys_procmem.o # name of syscall object file
```

After finishing your job, recompile and re-install kernel by steps in the next Section.

4 Compiling Linux Kernel

4.1 Build the configured kernel

Run "`make`" to compile the kernel and create `vmlinuz`. This step consumes a huge amount of time. To alleviate that issue, we can run this stage in parallel by using the tag "`-j np`", where `np` is the number of processes you want to 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 the memory by GRUB or other boot loaders that you use.

Then build the loadable kernel modules. In a like manner, 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`"?

4.2 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
```

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

```
sudo reboot
```

After logging in the computer again, run the following command:

```
uname -r
```

If the output string contains your Student ID then it means that your custom kernel has been compiled and installed successfully. You could proceed with the next step.

4.3 Testing

After booting to the new kernel, create a small C program to check if the system call has been integrated into the kernel.

Listing 7: Testing the system call

```
#include <sys/syscall.h>
#include <stdio.h>
#include <unistd.h>
#define SIZE   100

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

Remember to replace [Number_32] by the number of the procmem system call which was defined in the file syscall_32.tlb

After compiling and executing this program, your Student ID should be shown on the screen.

QUESTION: Why could this program indicate whether our system call works or not?

5 Wrapper

Although the procmem system call works properly, we still have to invoke it through its number which is quite inconvenient for programmers so we need to implement a C wrapper for it to make it easy to use. To avoid recompiling the kernel again, we leave the kernel source code directory and create another directory to store the source code for our wrapper. We first create a header file which contains procmem.h the definition of the wrapper and declare the proc_segs struct as follows:

Listing 8: procmem.h

```
#ifndef _PROC_MEM_H_
#define _PROC_MEM_H_
#include <unistd.h>
#include <sys/types.h>

struct proc_segs {
     unsigned long studentID;
     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 sys_procmem(pid_t pid, struct proc_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 do 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:

Listing 9: procmem.c

```
#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 in Listing 7.

6 Validation

You could validate your work by writing a test module to call the function `procmem`. After making sure that the wrapper works properly, we then install it to our virtual machine. First, we must ensure that everyone could access this function by making the header file visible to GCC. Run the following command to copy our header file to the header directory of our system:

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

QUESTION: Why is root privilege (e.g. adding sudo before the cp command) required to copy the header file to `/usr/include`?

We then compile our source code as a shared object to allow users to integrate our system call into their applications. To do so, run the following command:

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

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

We only have one last step: validate all of your work. To do so, write the following program, and compile it with the `-lprocmem` option. The result should be consistent with the content of the `/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 proc_segs info;

     if (procmem(mypid, &info) == 0) {
          printf("Student ID: %lu \n", info.studentID);
          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 its maps file
     // sleep(100);
}
```

7 Submission

7.1 Source code

After you have finished the assignment, you need to move all the following files into a single directory named as "StudentID" and compress this directory as **OSL2019_GROUP_STUDENTID_Assignment1.zip**

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

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

7.2 Report

Your report should comprise the following content:

- Introduction: Make a case for your work. Explain to the readers what problem did you solve.
- Methodology: What did you do? Elaborate on what you did to solve the problem. This is the most important part in your report. In developing this section, add your answers to questions denoted by the highlighted word "**QUESTION**" in this assignment.
- Results: What did you accomplish? Present the results of your work
- Discussion: What does it mean? Discuss the meaning of your results, do not leave the readers asking "so what?"

FYI: This is called as an IMRAD (Information, Methods, Results and Discussion) structure/format, you can lookup this term online.

Your report MUST be from 4 to 8 pages (font size 10-11pt), do NOT include a title page. The score of your report will be based on the correctness of your answers and the clarification of the content. Since this report accounts for a relatively high portion of your total score, you should spend much of your effort on it. There are some good report templates that you can use as a reference:

- https://www.overleaf.com/latex/examples/example-of-elsevier-article-template-with-dummy-text/qfscmwntknmq#.Wr-iXnVuY_4
- https://www.overleaf.com/latex/templates/template-for-submissions-to-scientific-reports/xyrztqvdccns#.Wr-iaXVuY_4
- https://www.overleaf.com/latex/templates/elsevier-article-template-with-different-bibliography-styles/npwqmwvhvvrk#.Wr-i9nVuY_4

7.3 Deadline and Grading

The deadline for BKeL submission is **May 22th, 2019**. After this deadline, you can still submit your report until May 25th, 2019 for LATE submission, but you will lose **20%** of your score.

Furthermore, since your results are graded based on your work on the VM built on your laptop. Please remember to bring your laptop for grading. Besides, in case you have finished early, you can schedule a meeting with me on campus show your achievement.

Best of luck !