HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY, VNU-HCM
FACULTY OF COMPUTER SCIENCE & ENGINEERNG

OPERATING SYSTEM

Assignment

System Call

Tutor: Pham Hoang Anh
Group: CC01
Student name: Tran Trung Quan - 1752044

Contents

1 Introduction

This report explains the steps to add a new system call that provides information about the memory layout of a given process. The strategy is modifying the Linux kernel to add our system call and deploying the kernel on the virtual machine. Toward the finish of the report, a new Linux kernel had compiled included the new system call and ready to deploy on any machine.

2 Methods

A brief about the stage to achieve the result:

- Preparing the virtual machine.
- Preparing the system call.
- Compiling kernels.
- Writing a wrapper for the system call.
- Validate the work.

More details about the work will be discuss below:

2.1 Preparing the virtual machine

Compiling and installing a new kernel is a risky task so working with the kernel on the virtual machine is needed. In this work we use the kernel version *4.4.56* work on *Ubuntu 16.04 LTS* virtual machine.

Install the core packages: Get Ubuntu's toolchain (gcc, make,...) and some necessary packages to compile the kernel.

```
$ sudo apt-get update
$ sudo apt-get install build-essential openssl libssl-dev vim
```

Install the *kernel-package*:

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

QUESTION: Why do we need to install kernel-package?

Answer: This package provides the capability to create a Debian kernel image package by just running make-kpkg kernel_image in a kernel source directory tree. In general, this package is very useful if you need to create a custom kernel, for example:

- Enable experimental features that are not part of the default kernel.
- Enable support for a new hardware that is not currently supported by the default kernel.
- Debug the kernel.
- Learn how kernel works, you might want to explore the kernel source code, and compile it on your own.

Create a kernel compilation directory and download the kernel source:

```
$ mkdir ~/kernelbuild && 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?

> *Answer:* The kernel cannot be compiled without a compiler, and the compiler cannot be run if there is no kernel (OS). So if you want to compile the kernel, you need the source and the compiler, run it on an OS. We can not just compile the local kernel on the running OS directly because it is handling the compiler and other processes.

Unpack the kernel source:

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

CONFIGURE THE NEW KERNEL:

Kernel configuration is set in its *.config* file, which includes the use of Kernel modules. By setting the options in *.config* properly, the kernel and computer will perform efficiency. Since making our own configuration file is a complicated process, we should reuse the content of the configuration file of an existing kernel currently used by the virtual machine:

```
$ cp /boot/config-4.4.0-21-generic ~/kernelbuild/linux-4.4.56/.config
```

Note: 4.4.0-21-generic is the version of the kernel installed in the virtual machine, checked by *uname -r* command.

Now we rename our new kernel to avoid overwriting one of our existing kernels:

```
$ vim ~/kernelbuild/linux-4.4.56/.config
// Change value of the line below
CONFIG_LOCALVERSION=".1752044"
// Save the file
```

2.2 Preparing the system call

We will prepare a syscall 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, we 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>*.
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>*.

IMPLEMENTATION

Modern processors support invoking system calls in many diﬁerent ways depending on the architecture. Since our virtual machine runs on x86 processors, we only concern Linux's system call implementation for this architecture.
Add our new system call:

```
$ vim ~/kernelbuild/linux-4.4.56/arch/x86/entry/syscalls/syscall_32.tbl
// Add the line below
377     i386     procmem     sys_procmem
// Save the file
```

```
$ vim ~/kernelbuild/linux-4.4.56/arch/x86/entry/syscalls/syscall_64.tbl
// Add the line below
546     x32     procmem     sys_procmem
// Save the file
```

337 and 546 are values 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?

Answer: The file syscall_32.tbl and syscall_64.tbl have the structure:
<number> <abi> <name> <entry point> <compat entry point>

<number>: the index number, the userspace requests a system call by specifying the index number of the syscall.
<abi>: Application Binary Interface, the interface between two modules, one of them is usually a library or operating system, usually code x64, x32 and i386.
<name>: name of the syscall.
<entry point> <compat entry point>: the kernel function name implementing the system call.

Add necessary information to kernel's header:

```
$ vim ~/kernelbuild/linux-4.4.56/include/linux/syscalls.h
```

Add the lines below:

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

QUESTION: What is the meaning of each line above?

Answer:
struct proc_segs: define a struct named proc_segs.
asmlinkage: This is a #define for some gcc magic that tells the compiler that the function should not expect to find any of its arguments in registers (a common optimization), but only on the CPU's stack. Recall our earlier assertion that system_call consumes its first argument, the system call number, and allows up to four more arguments that are passed along to the real system call. All system calls are marked with the asmlinkage tag, so they all look to the stack for arguments. It is also used to allow calling a function from assembly files.

Implement our system call:

```
$ touch ~/kernelbuild/linux-4.4.56/arch/x86/kernel/sys_procmem.c
$ vim ~/kernelbuild/linux-4.4.56/arch/x86/kernel/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)
{
  info->studentID = 1752044;
  struct task_struct *task;
  for_each_process(task)
  {
    if (task->pid == pid)
    {
      if (task->mm != NULL)
      {
        info->start_code = task->mm->start_code;
        info->end_code = task->mm->end_code;
        info->start_data = task->mm->start_data;
        info->end_data = task->mm->end_data;
        info->start_heap = task->mm->start_brk;
        info->end_heap = task->mm->brk;
        info->start_stack = task->mm->start_stack;
        return 0;
      }
    }
  }
  return -1;
}
```

We have to inform the compiler to include our new source file in the compilation process when we rebuild the kernel:

```
$ vim ~/kernelbuild/linux-4.4.56/arch/x86/kernel/Makefile
// Add the line below
obj-y += sys_procmem.o
```

2.3 Compiling Linux Kernel

Build the configured kernel

```
$ cd ~/kernelbuild/linux-4.4.56/
$ sudo make -j 4
```

Run "make" to compile the kernel and create vmlinuz that 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:

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

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

Answer:
make: compiles and links the kernel image. This is a single file named vmlinuz, this file is a compressed Linux kernel, and it is bootable.
make modules: just compile the modules, leaving the compiled binaries (the object files) in the build directory.

Installing the new kernel:
Install the modules:

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

Install the kernel itself:

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

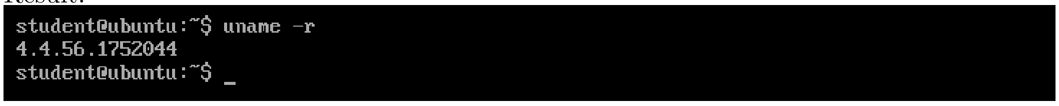
After installing then reboot:

```
$ sudo reboot
```

Verify if install successfully:

```
$ uname -r
```

Result:

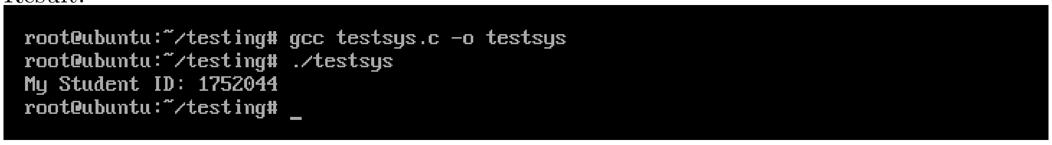


```
student@ubuntu:~$ uname -r
4.4.56.1752044
student@ubuntu:~$ _
```

Testing:
After booting to the new kernel, create a testsys.c to check if the system call has been integrated into the kernel.

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

int main()
{
  long sysvalue;
  unsigned long info[SIZE];
  sysvalue = syscall(377, 1, info);
  printf("My Student ID: %lu\n", info[0]);
}
```

Result:



```
root@ubuntu:~/testing# gcc testsys.c -o testsys
root@ubuntu:~/testing# ./testsys
My Student ID: 1752044
root@ubuntu:~/testing# _
```

2.4 Writing 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.
procmem.h:

```c
#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, st r u ct proc_segs * info)
# endif // _PROC_MEM_H_
```

QUESTION: Why do we have to re-define proc_segs struct while we have already defined it inside the kernel?

> *Answer:*
> The proc_segs struct that we defined inside the kernel is used in kernel space we must re-define it if we want to use in the userspace.

procmem.c to hold the source code file for wrapper:

```
#include "procmem.h"
#include <linux/kernel.h>
#include <sys/syscall.h>
#define _SYS_PROCMEM_ 377

long procmem(pid_t pid, struct proc_segs *info)
{
  long sysvalue;
  sysvalue = syscall(_SYS_PROCMEM_, pid, info);
  return sysvalue;
}
```

2.5 Validation

Copy our header file to the header directory:

```
$ 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?

> *Answer:*
> We are in the "student" user account that only has the user privileges under /home/student/. /usr/ is owned by the root account so to write files in there you need to write them as root (using sudo).

Compile our source code as a shared object to allow users to integrate our system call into their applications:

```
$ gcc -shared -fpic procmem.c -o libprocmem.so
$ sudo cp <path to libprocmem.so> /usr/lib
```

QUESTION: Why must we put -shared and -fpic options into the gcc command?

> *Answer:*
> *-shared:* Produce a shared object which can then be linked with other objects to form an executable.
> *-fpic:* Position Independent Code (PIC) means that the generated machine code is not dependent on being located at a specific address in order to work.
> Whenever a shared lib is loaded, the loader (the code on the OS which load any program you

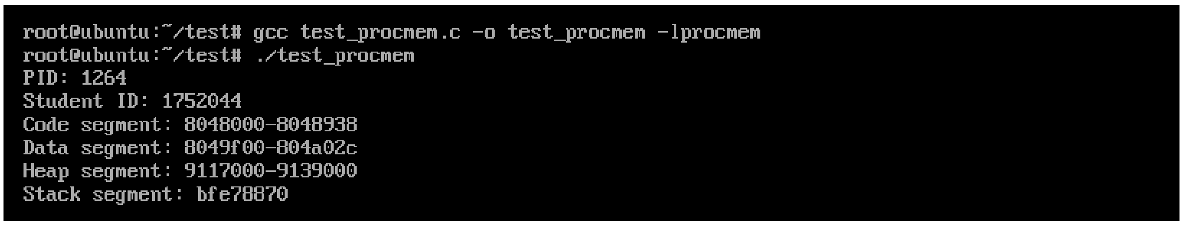
run) changes some addresses in the code depending on where the object was loaded to. Code that is built into shared libraries should normally be position-independent code (PIC), so that the shared library can readily be loaded at (more or less) any address in memory.

3 Results

Write a small program to test our work:

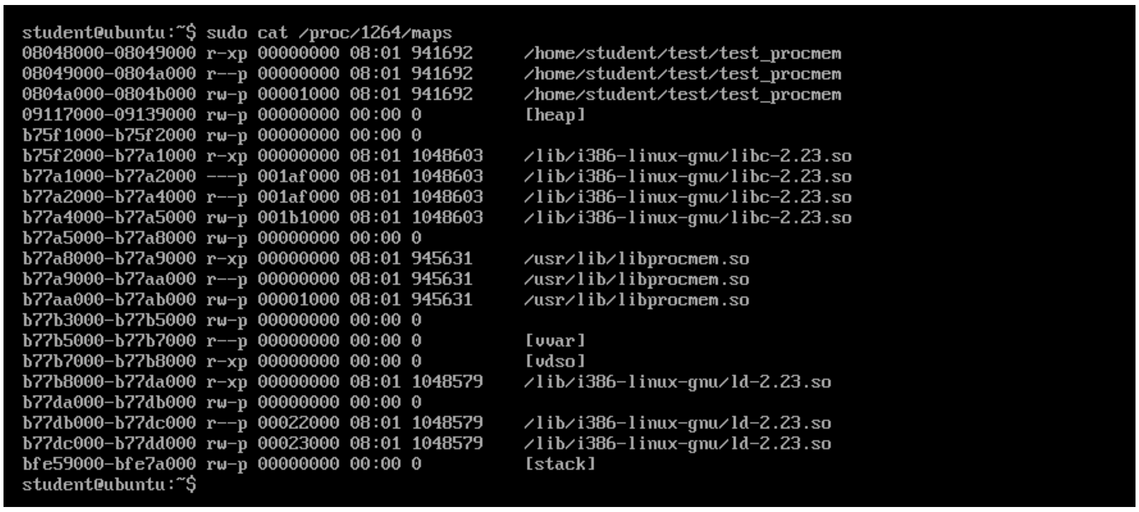
```c
#include <procmem.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdint.h>

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);
  }
    // sleep(100);
}
```

Compiled with -lprocmem and here is the result:



```
root@ubuntu:~/test# gcc test_procmem.c -o test_procmem -lprocmem
root@ubuntu:~/test# ./test_procmem
PID: 1264
Student ID: 1752044
Code segment: 8048000-8048938
Data segment: 8049f00-804a02c
Heap segment: 9117000-9139000
Stack segment: bfe78870
```

Check again with maps:



```
student@ubuntu:~$ sudo cat /proc/1264/maps
08048000-08049000 r-xp 00000000 08:01 941692     /home/student/test/test_procmem
08049000-0804a000 r--p 00000000 08:01 941692     /home/student/test/test_procmem
0804a000-0804b000 rw-p 00001000 08:01 941692     /home/student/test/test_procmem
09117000-09139000 rw-p 00000000 00:00 0          [heap]
b75f1000-b75f2000 rw-p 00000000 00:00 0
b75f2000-b77a1000 r-xp 00000000 08:01 1048603    /lib/i386-linux-gnu/libc-2.23.so
b77a1000-b77a2000 ---p 001af000 08:01 1048603    /lib/i386-linux-gnu/libc-2.23.so
b77a2000-b77a4000 r--p 001af000 08:01 1048603    /lib/i386-linux-gnu/libc-2.23.so
b77a4000-b77a5000 rw-p 001b1000 08:01 1048603    /lib/i386-linux-gnu/libc-2.23.so
b77a5000-b77a8000 rw-p 00000000 00:00 0
b77a8000-b77a9000 r-xp 00000000 08:01 945631     /usr/lib/libprocmem.so
b77a9000-b77aa000 r--p 00000000 08:01 945631     /usr/lib/libprocmem.so
b77aa000-b77ab000 rw-p 00001000 08:01 945631     /usr/lib/libprocmem.so
b77b3000-b77b5000 rw-p 00000000 00:00 0
b77b5000-b77b7000 r--p 00000000 00:00 0          [vvar]
b77b7000-b77b8000 r-xp 00000000 00:00 0          [vdso]
b77b8000-b77da000 r-xp 00000000 08:01 1048579    /lib/i386-linux-gnu/ld-2.23.so
b77da000-b77db000 rw-p 00000000 00:00 0
b77db000-b77dc000 r--p 00022000 08:01 1048579    /lib/i386-linux-gnu/ld-2.23.so
b77dc000-b77dd000 rw-p 00023000 08:01 1048579    /lib/i386-linux-gnu/ld-2.23.so
bfe59000-bfe7a000 rw-p 00000000 00:00 0          [stack]
student@ubuntu:~$
```

Now we can compress the linux-4.4.56 folder, rename it and distribute to any machine to install the kernel included our new system call.

4 Discussion

In computing, a system call is how a program requests a service from an operating system's kernel. This may include hardware related services, creating and executing new processes, and communicating with integral kernel services. System calls provide an essential interface between a process and the operating system.

The design of the microprocessor architecture on practically all modern systems (except some embedded systems) involves a security model which specifies multiple privilege levels under which software may be executed; for instance, a program is usually limited to its own address space so that it cannot access or modify other running programs or the operating system itself, and a program is usually prevented from directly manipulating hardware devices.

So develop a system call help us to control the hardware that implemented into the operating system with different privileges. Prevent illegal access and changing the hardware from user processes.