



OPERATING SYSTEM

Assignment - SYSTEM CALL

Student: Phạm Trọng Nhân - 1752394

Ho Chi Minh, 05/2019

Introduction

From this assignment, we need to develop a function which can retrieve information about memory layout for specific program. In that case, we will develop a system call **procmem** which invokes with built-in struct `task_struct` to get memory layout from kernel mode. And finally, we will learn to resolve 3 problems:

- How to get memory layout from `task_struct`.
- And how to send data from kernel mode to user mode.
- Add new system call to kernel and wrapper for that function.

Methodology - Configuration - Implementation

Prepare Linux Kernel

Because Linux kernel is written in C language, so we must install **build-essential** and **kernel-package**. From here there are 2 scenario can happen:

- Base on your kernel in current running OS, if it too old and replace by a newer one in update state. There will be a prompt to keep or change `kernel-img.conf`.
- As instruction, we will continue to use some information from configuration files, so it's important to keep it as previous.

1. Why we need to install **kernel-package** ?

kernel-package is a package for automate the routine steps to compile and install a custom kernel. And also for very important reason, it allows us to keep multiple version of kernel images on our machine with no fuss (Here are original and linux-4.4.56).

2. 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?

We always can use our original kernel on running OS because that kernel can run mostly on any system configuration. But however, Debian-based OS may include or change some part in their kernel, so you should make sure this kernel is clean. Besides that, another kernel source from <http://www.kernel.org> give us some advantages.

First, we can use some options and configuration that running kernel may not include in or unnecessary. This is very important because it will support for almost any hardware configuration. Second, give the better optimization compare to kernel in running OS because contains less unnecessary system functions. Similar to previous reason, the difference in the number of system call in kernel give the different compatibility and way to implement new system calls.

Third, have the latest support from providers and supporters. Newer kernel have the better performance compare to its older versions and have lesser troubles and warnings while compiling kernel. Finally, creating a monolithic kernel instead of modularized one. This is based on running OS kernel, some of OS run modularized kernel which is not suitable to learn how to implement new system calls.

Configuration

After installing enough and required package, we follow instruction and start to configurate **.config** to modify Local version ID of our kernel.

To configurate **.config** file, we have 2 options as mentioned, can modify through **make** command or directly into **.config** file. But now we can't see the result, to see if it works or not, we must wait until installing this kernel.

Implementation

After learning defined struct which will useful in system call and modify Local version. It's time to write system call and must follow these steps:

- Declare system call in system call table both in 32-bit and 64-bit OS. Remember number you just created in system call 32-bit table, we'll use later.
- Write and self-evaluate the correctness of new system call.
- Add new command for new system call file into Makefile to make sure it doesn't mis-compiling later.

Follow these steps as instruction, we can finish create a new system call. But there are some reminder to create new system call easier.

- Must read document about struct **task_struct** and **mm_struct** to know fully what they contain.
- To get correct memory layout of process by PID, it is required to use **for_each_process** loop.

3. When add a new system call in `syscall_32.tbl`, what is the meaning of other components, i.e. `i386`, `procmem`, and `sys__procmem` ?

In this steps, we're configuring 2 files which are system call table. For 32-bit system, we only need to change in `syscall_32.tbl` but to make sure it runs perfectly on machine we should add in `syscall_64.tbl` too. So in each table, the first line is variable name for each column:

- **i386** is in **abi** - Application Binary Interface. For different binaries to interact, they need to agree on a set of interfaces. Here with `i386` in `syscall_32.tbl`, that means for compatibility with x86 32-bit binaries. While in `syscall_64.tbl` is `x32` for ILP32 (32-bit int, long, pointers).
- **procmem** is **name**, Linux system provides headers with system call number. Here **name** is a macro name - wrapper for that system call.
- **sys__procmem** is **entry point**, is a function name in kernel space to run the system call.

4. What is the meaning of each new lines adding to `include/linux/syscalls.h`

The **syscalls.h** is header file of all system calls in this kernel. So before implementing a new system call, we need to declare it in the header file. The same reason with **struct proc_segs**, this struct is important for retrieving information which we will each in this system call. The further more, we will declare variables in this struct.

Another important point in lines are **asm__linkage**, this tag is a `#define` to tell compiler that the function should not expect to find any of its arguments in registers (a common optimization), but only on the stack. So this is more related to **syscall** which take the system call number and 4 more arguments.

So it will leave its other arguments on the stack. And now all system calls with `asm linkage` will look its arguments on stack. It is also used to allow calling a function from assembly files.

And the rest of system call declaration are **pid** of process we want to know memory layout and struct **proc_segs** which to store it.

Compiling Linux kernel

This part is the most time-consuming when creating new system call. Must compile all source code and modules for kernel. This work consist of 2 step, but the problem is that if your new system call have errors the compiler may or may not stop. So it's careful to check for output object file.

After compiling the kernel and creating `vmlinux` image without errors, we can install this kernel image to current OS. I suggest that before installing new kernel, we should create a snapshot of current state because there are some error might happen after wrong installation.

5. What is the meaning of 2 states "make" and "make modules"

In this part, we do 2 works about compiling files and linking. But there are a lot of reason behind that work and what they do.

Firstly, command **make** did as all normal Makefile did, it will compiles and links the kernel image, this image is **vmlinux** as described in guideline. So this **make** command will compiles all source code files in system calls and driver which are currently in kernel. And with our new system call, we also add it to Makefile like previous step. This step is quiet difference from normal where all source code files is compiled into object files.

The second command **make modules**, this command in Makefile compiles individual files which is marked. The mark signature, you can view in compiling stage, there are a lots lines which have **[M]** between GCC and code files. Those outputs will be object files which link against to our freshly-build kernel.

The important reason after 2 steps are the result, after compiling and linking, we will have **vmlinux** and **bzImage** are important part for installing new kernel.

6. Why the testing code can indicate whether our system call work or not ?

The testing code may not actually show the system call work or not. The result we can have in the testing code is only the student ID. So we can only know sure that from the system call we can retrieve information in kernel space to user space. But can not conclude that the information about memory layout is correct for specific program (using PID) we want.

In conclusion, this testing code can indicate whether system call can transfer data from kernel space to user space, if it can so this system call works.

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. This work consists of header file and source code, the result we expect is a shared object file can work as a new options in GCC command. Follow the instruction to create header and source code files.

In source code file, we just implement again syscall function with 2 arguments as we wrote in testing code.

7. Why we have to re-define proc_segs struct while we have already defined it inside the kernel ?

In Linux system calls are functions which can interact with both kernel and user space. From the beginning until now, we implemented the system call in kernel space only, which are included declaring new data struct and system call.

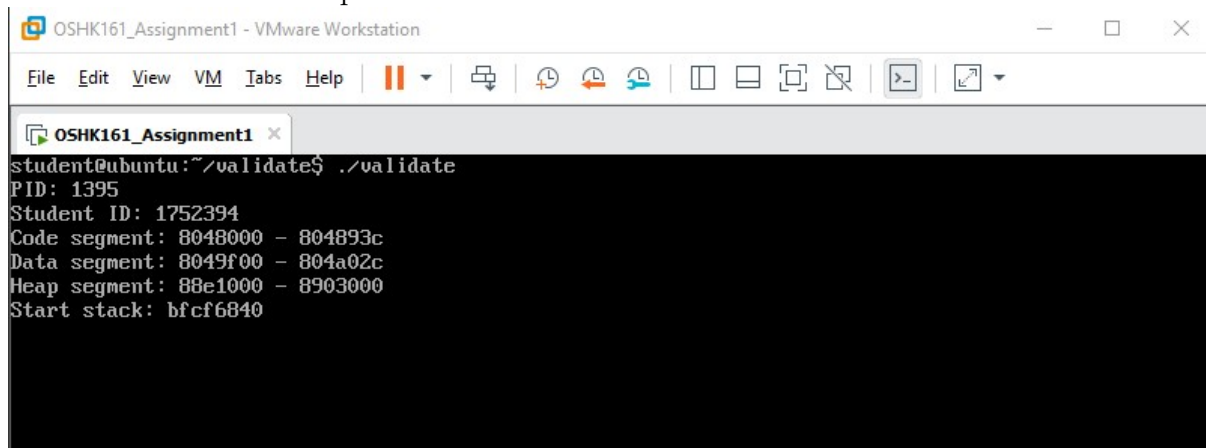
So our wrapper is currently declared and used in user space, can't access to the kernel space to use defined struct. So this is case, we must define again proc_segs to use in this wrapper and let user use.

Validation

After finishing create a wrapper for new system call, to make it useable as default function, we must copy the header and object files to right place. Done with compiling and copy new file, we can test again this wrapper (system call) with ease and try to print out all information in proc_segs struct, to test the correctness of values.

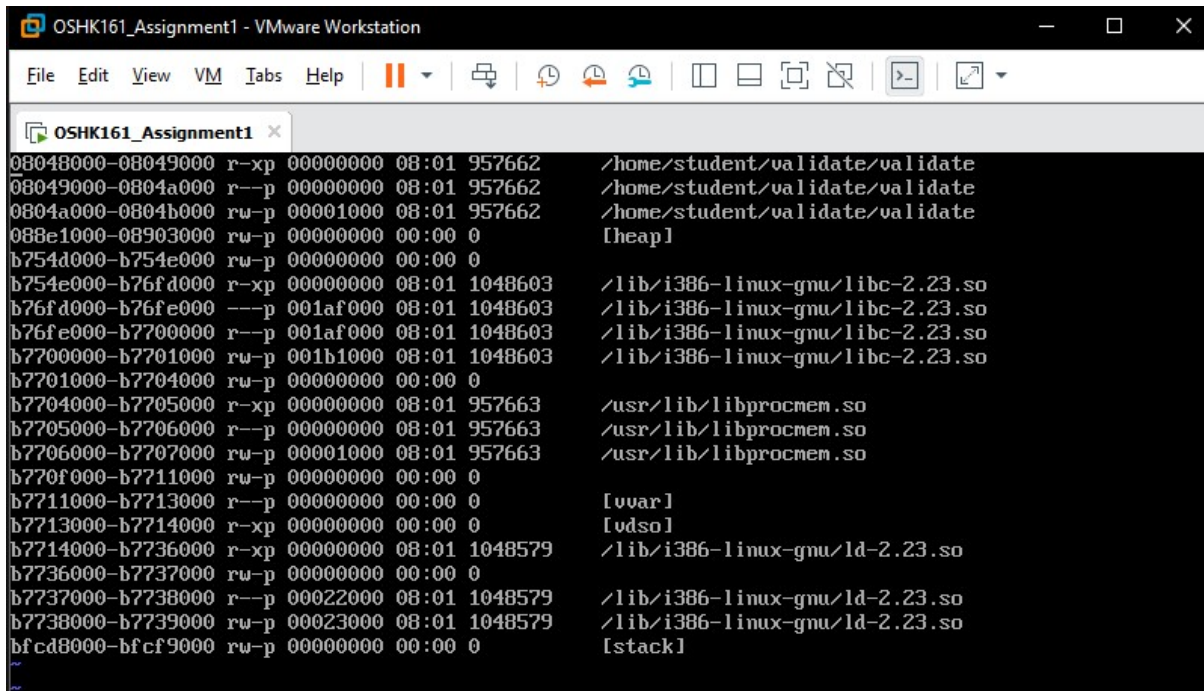
If everything work fine, you should create a snapshot of current state in case there are any changes you make later on.

The result we can expect are similar to these:



```
OSHK161_Assignment1 - VMware Workstation
File Edit View VM Tabs Help
OSHK161_Assignment1 x
student@ubuntu:~/validate$ ./validate
PID: 1395
Student ID: 1752394
Code segment: 8048000 - 804893c
Data segment: 8049f00 - 804a02c
Heap segment: 88e1000 - 8903000
Start stack: bfcf6840
```

Result from validation code



```

OSHK161_Assignment1 - VMware Workstation
File Edit View VM Tabs Help
OSHK161_Assignment1 x
08048000-08049000 r-xp 00000000 08:01 957662 /home/student/validate/validate
08049000-0804a000 r--p 00000000 08:01 957662 /home/student/validate/validate
0804a000-0804b000 rw-p 00001000 08:01 957662 /home/student/validate/validate
088e1000-08903000 rw-p 00000000 00:00 0 [heap]
b754d000-b754e000 rw-p 00000000 00:00 0
b754e000-b76fd000 r-xp 00000000 08:01 1048603 /lib/i386-linux-gnu/libc-2.23.so
b76fd000-b76fe000 ---p 001af000 08:01 1048603 /lib/i386-linux-gnu/libc-2.23.so
b76fe000-b7700000 r--p 001af000 08:01 1048603 /lib/i386-linux-gnu/libc-2.23.so
b7700000-b7701000 rw-p 001b1000 08:01 1048603 /lib/i386-linux-gnu/libc-2.23.so
b7701000-b7704000 rw-p 00000000 00:00 0
b7704000-b7705000 r-xp 00000000 08:01 957663 /usr/lib/libprocmem.so
b7705000-b7706000 r--p 00000000 08:01 957663 /usr/lib/libprocmem.so
b7706000-b7707000 rw-p 00001000 08:01 957663 /usr/lib/libprocmem.so
b770f000-b7711000 rw-p 00000000 00:00 0
b7711000-b7713000 r--p 00000000 00:00 0 [vvar]
b7713000-b7714000 r-xp 00000000 00:00 0 [vdso]
b7714000-b7736000 r-xp 00000000 08:01 1048579 /lib/i386-linux-gnu/ld-2.23.so
b7736000-b7737000 rw-p 00000000 00:00 0
b7737000-b7738000 r--p 00022000 08:01 1048579 /lib/i386-linux-gnu/ld-2.23.so
b7738000-b7739000 rw-p 00023000 08:01 1048579 /lib/i386-linux-gnu/ld-2.23.so
bfc48000-bfc49000 rw-p 00000000 00:00 0 [stack]

```

Result from `/proc/<pid>/maps`

When comparing result in both output and maps, we notice some difference and have conclusion:

- Memory is divided into blocks of 4096 bytes (4 KB). So in maps file we can see that memory of each segments use entire or part of a block
- Heap memory location is the same with start and end point.
- Code segment memory doesn't use entire a block of memory. But because of memory alignment, then no other segments interfere to its memory.
- Data segment memory should use next blocks memory, but result in validate code shows that data segment start in the middle of block. That being said, the big memory between code and data segment is used for something else which didn't retrieve from `task_struct`.
- And finally, Stack memory of process is allocated in the middle of the block.
- Beside information of memory address in maps, there are others for each segments we should notice in order:
 - address: The starting and ending address of the region in the process's address space.
 - permissions: Which action can perform on this region address, consists of read, write, execute and shareable.
 - offset: If the region was mapped from a file, this is the offset in the file where the mapping begins. If the memory was not mapped from a file, it's just 0.
 - device: If the region was mapped from a file, this is the major and minor device number where file lives.
 - inode: This is file number if the region mapped from file.
 - pathname: This is name of the file name or name of segment (like heap or stack).

8. Why is root privilege required to copy the header file to `/usr/include` ?

The **user** directory is a part of root directory of root account. This root directory is the top level directory on any Unix-like OS (contains all directories and their subdirectories). It is designated by a forward slash (/) and its subdirectories are **bin**, **boot**, **dev**, **etc**, **home**, **mnt**, **sbin** and **usr**.

So to access into this directory, root account is required (which is also referred to as the root user) to copy header file.

9. Why must we put `-shared` and `-fpic` options into the gcc command ?

To understand these options, we should investigate their usage in GCC document:

- **-shared**: The output file from gcc command with this option produces a shared object which can then be linked with other objects to form a executable. So from definition, this option is only start working in Linker step in compiling.
- **-fpic**: Generate position-independent code suitable for use in a shared library.

So from definition, we can understand why add 2 options, we are developing a wrapper for a system call which will be used in entire system. It is important to have ability to use this function directly like other C default libraries. To do that, result object file will be in `/usr/lib`.

But this is a system call so to create output file from source code file which used this system call, the GCC command to compile that code should have options to link to correct wrapper. This is where 2 options handy, the object file we created is `.so` type which is object file for header also a new option in GCC command.

Conclusion

After all work, we can have a strong foundation and knowledge to create a system call in Linux kernel which help us know how Linux system should work. This knowledge also consists of know how memory allocation work in kernel by giving us a way to read memory layout of specific process running.

And base on our new system call, we have deeper understanding of memory blocks and file arrangement in kernel. On comparison between system call result and maps, we know that memory of segments in the same process not exactly allocated next to each others continuously because of memory region and memory alignment.

The knowledge of memory alignment will us evaluate how much a program actual uses instead of raw calculate of type's size.