Add Your Own System Calls to the Linux Kernel

williamthegrey.wordpress.com/2014/05/18/add-your-own-system-calls-to-the-linux-kernel/

williamthegrey May 17, 2014

Environment

- OS: Fedora 19 64-Bit
- Target Kernel: linux-3.13.6-100.fc19.x86_64

Introduction

It should have been easy to add user-defined system calls to the Linux kernel. But since a version of kernel which I don't exactly know, the method to accomplish this is quite different from what you did with old kernels like version 2.6.x due to some changes during the time.

Following is how you will deal with the new kernels like version 3.13.6 which I used here.

Get the Linux kernel Source

There two types of kernel. The one is the kernel you get from The Linux Kernel Archives. This kind of kernel is sometimes called the "vanilla" kernel. The other one is the kernel you get from the website of the Linux Distribution you are using. For the latter, go to Ubuntu's website if you use Ubuntu, or go to Fedora's website if you use Fedora, and so on. Here I used the kernel downloaded from Fedora's website, which is linux-3.13.6-100.fc19.x86_64.

By the way, I never fully understood the exact differences between the two types of kernels.

Set the System Call Number

Edit "arch/x86/syscalls/syscall_64.tbl". Locate the entries where `common` abi is always used, and add another entry right below them. Like this:

```
312     common  kcmp                    sys_kcmp
313     common  finit_module            sys_finit_module
314     common  my_operation            sys_my_operation
```

The last row is what I added. `314` is the system call number I used. You should determine your own number which should be 1 greater than the number right above your entry. `my_operation` is the system call name I defined here, and `sys_my_operation` is the entry point.

Declare the System Call Function Prototype (Entry Point)

Edit "include/linux/syscalls.h", and add your function prototype before `#endif` like this:

```
asmlinkage long sys_my_operation(int x, int y, int* result, int rule);
#endif
```

`sys_my_operation` is the entry point defined in “arch/x86/syscalls/syscall_64.tbl”.

Implement the System Call Function

Edit “kernel/sys.c”, and choose a `SYSCALL_DEFINE` macro to implement your system call function at the end of this file like this:

```
SYSCALL_DEFINE4(my_operation, int, x, int, y, int*, result, int, rule)
{
        long error = 0;
        printk("sys_my_operation : sys_my_operation is called.\n");

        ...

        return error;
}
```

`SYSCALL_DEFINE4` macro is used because this system call has 4 parameters according to my function prototype declared in “include/linux/syscalls.h”. You should use an appropriate macro according to your own function prototype.

The parameters of the macro are the system call name, the parameter types and the parameter names of your system call. `sys_my_operation` is the system call name. `int` is the parameter type of the first parameter, and `x` is the parameter name of the first parameter, and so on.

`printk` function is used here to output a line of message from which we can see whether this system call is called.

Build the Kernel

Build your customized kernel using the commands below:

```
make mrproper
make menuconfig
make bzImage
make modules
sudo make modules_install
sudo make install
```

Then reboot your machine and choose your new kernel in grub. Now you are using your new kernel with your own system call in it.

Test Your System Call

Write a user program to check if your system call works. The code of this program may look like this:

```
#include <unistd.h>
...

int main(int argc, char *argv[])
{
 int x, y, result, rule;
 long error;

 ...

 error = syscall(314, x, y, &result, rule);

 ...

 return 0;
}
```

Call your system call by using `syscall` function. The first parameter `314` is the system call number of your system call. You should use your own value. The rest of the parameters of `syscall` function are the parameters of your system call. `syscall` returns 0 on success, or -1 on failure. You may want to use `errno` to determine the problem.

After running the user program, you can use the following command to get the messages printed by `printk` function in the system call:

```
sudo dmesg -c
```

In addition, you can use `strace` command to trace all the system calls and signals during the user program's running.