Principles of Operating Systems

CMSC 421 - Spring 2018

Adding a System Call to the Linux Kernel (Version 4.15.0)

Introduction

Project 1 in this course will involve adding several system calls to add interesting functionality to the Linux kernel. In preparation for this task, this guide has been developed to show you how to go about adding a simple system call to the kernel. This guide will focus on the 64-bit x86 architecture, however similar steps could be taken on other CPU architectures that are supported by the Linux kernel.

The system call added to the kernel in this guide is basically pointless, but it demonstrates the process in a relatively simple manner. All the system call developed in this guide does is print a simple "Hello World!" string to the kernel's log.

System Call Implementation

To start out, make a new directory in the root of your kernel source tree. For the purposes of this example, let's just call the directory `hello`. Create a new file in that directory called `hello.c`.

In the `hello.c` file created above, let's implement the system call itself. Copy/paste the following code into the source file...

Source Code:

```
    #include <linux/kernel.h>

    asmlinkage long sys_hello(void) {
        printk("Hello World!\n");
        return 0;
    }
```

As mentioned before, this system call doesn't do anything particularly interesting. All it does is print the string "Hello World!" to the kernel's message log. We do not have a direct `printf` to use within kernel space. While it would be possible to use the `write` system call to print to the calling process' `stdout`, I'll leave that as an exercise to the reader.

There are a couple of things to notice about this system call code. First of all, you should note that it returns 0 in all cases (as a long). This notifies the calling process that the system call succeeded and that no errors occurred. In fact, any positive return value from a system call is treated as a successful return. If the system call were to indicate an error occurred, a negative number should be returned — specifically a negated value from the <errno.h> header file should be returned, indicating what type of error occurred. The system C library will automatically take that negative value and put the appropriate value in the `errno` variable in user-space (and return -1 from the function). Another point to notice is that the system call has the `asmlinkage` tag in front of the return value. This is required for system calls on some architectures, so you should not leave it out! Technically it isn't strictly required on some architectures, but it still doesn't hurt to add it in.

After creating the system call, we must create a `Makefile` in the syscall's directory to build the code when the kernel is built. Since the kernel has a relatively nice build system, the `Makefile` is extremely short and simple. Copy and paste the following into a new file named `Makefile` in the `hello` directory:

Source Code:

```
obj-y := hello.o
```

With that, we have completed all of the "new" code that needs to be added to the kernel.

Kernel Modifications for the New System Call

To ensure that our new system call code gets compiled, we must first add in the new `hello` directory to the normal kernel build system. To do this, open up the root level Makefile. Look for the following section of the Makefile (which should start at line 942 on a vanilla copy of the 4.15.0 kernel source code):

Source Code:

```
ifeq ($(KBUILD_EXTMOD),)
core-y          += kernel/ certs/ mm/ fs/ ipc/ security/ crypto/ block/

vmlinux-dirs    := $(patsubst %/,%,$(filter %/, $(init-y) $(init-m) \
                     $(core-y) $(core-m) $(drivers-y) $(drivers-m) \
                     $(net-y) $(net-m) $(libs-y) $(libs-m) $(virt-y)))
```

You must modify the `core-y` line to add our new directory. Modify it to look like this:

Source Code:

```
core-y          += kernel/ certs/ mm/ fs/ ipc/ security/ crypto/ block/ hello/
```

Next, open up the `include/linux/syscalls.h` file. We must modify this file to provide the prototype of the system call we've added. This would be needed, for instance, if we were to need to call the system call from other kernel code or system calls. It isn't strictly necessary for this particular system call, but it is still a good idea. Go to the bottom of the file and add the following right before the `#endif`:

Source Code:

```
asmlinkage long sys_hello(void);
```

Finally, we must add the new system call to the system call table. Open the `arch/x86/entry/syscalls/syscall_64.tbl` file, and go to the part of the file right before the beginning of the "x32-specific system call numbers" (which should be line 342 of the vanilla 4.15.0 kernel), and add the following after the `statx` line:

Source Code:

```
333     common  hello                   sys_hello
```

In this file, the first column represents the system call number. The second column indicates which ABI the system call is a part of. For x86-64, your options here are "common", "64", and "x32". Most of the time, you want to use the "common" option to make it so the system call will work regardless of which mode is currently in use. The third column is the descriptive name of the system call. The last column is the name of the actual function within the kernel that implements the system call. Generally, the fourth column will start with "sys_", but this is not strictly necessary. All but the second column in the file should have unique values for each entry.

If you are using a different kernel version, you may need to adjust that line slightly to account for any new/removed system calls in relation to version 4.15.0. Simply put, take the number on the line before and increment it by one to add the new system call. Make a note of the number of the system call, as you will need it later.

You have now finished modifying the kernel. All that's left now is to build and install your new kernel, reboot into it, and test it out. I'll leave the building, installing, and rebooting steps as an exercise to the reader here.

Testing Your New System Call

To test out your new system call, you must build a user-space program to call it. Create a new file (in your home directory) and copy/paste the following content into it:

Source Code:

```
#include <stdio.h>
#include <unistd.h>
#include <linux/kernel.h>
#include <sys/syscall.h>

#define __NR_hello 333

long hello_syscall(void) {
    return syscall(__NR_hello);
}

int main(int argc, char *argv[]) {
    hello_syscall();
    return 0;
}
```

Note that if you had a different number for your system call in the previous step (i.e, you're not using a vanilla copy of the 4.15.0 kernel source), then you will need to adjust the `__NR_hello` line. The only interesting part of this code is the part where the system call is called. This is done by making use of the `syscall` macro, provided by the C library. This function-like macro accepts an integer argument to note what system call to perform, along with any number of arguments to pass to the system call. So, if the hello system call accepted an argument (with a name, for instance), you would simply add that argument to the list passed there, after `__NR_hello`.

Compile the program with your system C compiler (`gcc`) and run it. You will not see any output if the program was successful. However, if you view the kernel's message log by running the `dmesg` program, you should see a "Hello World!" message at/near the end of the log. If so, congratulations, you've completed this little tutorial!

Further exploration

The system call developed in this tutorial is rather boring and doesn't really do anything of major interest. There are ways this could be improved. For instance, you could use the `write` system call within the kernel to write the greeting string to the user-space program's standard output. Another, more interesting modification would be to modify the system call to accept a string to greet the user by name. This would complicate the system call quite a bit, as you would have to handle proper copying of the argument into kernel-space, however it would make a very worthwhile exercise for anyone who wants to develop a non-trivial system call.
