Guide to Adding A SysCall to Linux 3.19+[1]

by Brian Fraser

Last update: April 7, 2015

This document guides the user through:

1. Adding a new system call (sys-call) to Linux (as of 3.19.1)
2. Writing a user-level program to call it.

Table of Contents

Formatting

1. Commands starting with `$` are Linux console commands on the host OS:
   ```
   $ echo Hello world!
   ```
2. Commands starting with `#` are Linux commands on the QEMU target VM:
   ```
   # echo Hello QEMU world!
   ```
3. Almost all commands are case sensitive.

It is assumed that the user has downloaded the Linux kernel source code, compiled it, and is able to boot the kernel. See "Custom Kernel Guide" on course website (see Further Resources for link).

Revision History

- March 30: Added some troubleshooting steps for if the sys-calls fail.
- April 7th: Added support for 32-bit systems.

[1]Based on guide created by Arrvindh Shriraman.

1. Adding a HelloWorld System Call

This guide assumes you have the Linux source code (for version 3.19.1, or more recent may also work) in a folder `linux-stable/`. **This guide focuses on 64-bit version of Linux, but gives hints for working with 32-bit.**

1. Verify if your host OS is 32-bit or 64-bit:
   ```
   $ uname -m
   ```
   - If output is:
     `i686`: 32-bit system.
     `x86_64`: 64-bit system.
2. Change to the `linux-stable/` folder
   ```
   $ cd linux-stable
   ```
3. Create a new folder named `cs300/` inside `linux-stable/`
   ```
   $ mkdir cs300
   ```
4. Create a new file which implements your new system call:
   ```
   $ gedit cs300/cs300_test.c
   ```
   - Set file contents to:
   ```
   #include <linux/kernel.h>

   // Implement a HelloWorld system call
   // Argument is passed from call in user space.
   asmlinkage long sys_cs300_test(int argument)
   {
         long result = 0;

         printk("Hello World!\n");
         printk("--syscall argument %d\n", argument);

         result = argument + 1;
         printk("--returning %d + 1 = %ld\n", argument, result);
         return result;
   }
   ```
   - This code will be compiled inside the kernel, not as a user level program. It will run with the privilege of the kernel and without the support of the standard C library.
   - `printk()` is the kernel's version of `printf()`. It has limited formatting capabilities compared to `printf()`.
   - The kernel is complied using the C90 standard, not C99. Therefore you must declare all your variables at the top of a block (such as your function) instead of in the middle (as permitted in C99). Hint: Always initialize your variables to *some* value!
5. Create a Makefile to allow your new system call file to be compiled by the kernel.
   ```
   $ gedit cs300/Makefile
   ```
   - Set file contents to just:
   ```
   obj-y := cs300_test.o
   ```
   - If adding additional .c files later, you can space separate them, such as:
   ```
   obj-y := cs300_test.o mytest.o otherthing.o
   ```

6. Integrate your new folder into the over-all kernel build process by editing the Linux kernel's main Makefile (in `linux-stable/`):
   ```
   $ gedit Makefile
   ```
   - Find the line which defines `core-y` (near line ~883)
     ```
     ifeq ($(KBUILD_EXTMOD),)
     core-y          += kernel/ mm/ fs/ ipc/ security/ crypto/ block/
     ```
   - Add your new folder to the end of the `core-y` define:
     ```
     ifeq ($(KBUILD_EXTMOD),)
     core-y          += kernel/ mm/ fs/ ipc/ security/ crypto/ block/ cs300/
     ```
   - Later when you make the kernel, it will also build the contents of your `cs300/` folder.
7. Actually create the new sys-call by adding it to the kernel's list of system calls.

   On a 64 bit system:

   - Open the file which creates all the sys-calls for the x86-64 bit architecture:
     ```
     $ gedit arch/x86/entry/syscalls/syscall_64.tbl
     ```
   - Add the following line at the end of the first block of sys-call defines (i.e., after the call numbered ~332, and before the second block defining calls in the 512+ range).
     ```
     340   common      cs300_test       sys_cs300_test
     ```
     - The number on the left is the sys-call number you are creating. This sys-call will exist only in your kernel. In general, once a sys-call is added to the main-line kernel that number is never reused because it would break any existing application which depends on it. For this guide, all that is important is that the number you select is not used by other sys-calls (above it).
     - `common` defines the application binary interface.
     - `cs300_test` is name of the sys-call, as will be listed in the header files in the kernel.
     - `sys_cs300_test` is the name of the function (the "entry point") which be called to service this sys-call. This matches the name of the function created in `cs300_test.c`.

On a 32 bit system:

   - Open the file which creates all the sys-calls for the x86-32 bit architecture:
     ```
     $ gedit arch/x86/entry/syscalls/syscall_32.tbl
     ```
   - Add the following line at the end of the file:
     ```
     390   i386        cs300_test       sys_cs300_test
     ```
     - See above for an explanation of these values.
8. Rebuild the kernel. The compiled kernel will now include your custom sys-call! Now all you have to do is write some code which calls it (next section).
   ```
   $ make -j4
   ```

2. Calling your System Call

2.1 Creating a Test Application

1. Outside of the `linux-stable/` folder (i.e., likely move to `~cmpt300/`) create a new folder for your user-level test application:
   ```
   $ cd ~                          – In the lab, use the /usr/shared/...../ folder
   $ cd cmpt300                    – Likely created while following Custom Kernel Guide
   $ mkdir test-syscall
   $ cd test-syscall
   ```
2. Create a test application:
   ```
   $ gedit cs300_testapp.c
   ```
   - Set contents to:
   ```
   #include <stdio.h>
   #include <unistd.h>
   #include <sys/syscall.h>

   #define _CS300_TEST_ 340     // for a 64 bit system
   //#define _CS300_TEST_ 390   // for a 32 bit system

   int main(int argc, char *argv[])
   {
         printf("\nDiving to kernel level\n\n");
         int result = syscall(_CS300_TEST_, 12345);
         printf("\nRising to user level w/ result = %d\n\n", result);

         return 0;
   }
   ```
   - `_CS300_TEST_` is defined to be the sys-call number we crated. Normally this would be imported in the `sys/syscall.h` file. However, since we are building a custom kernel, we can `#define` our custom syscall number and not have to worry about updating .h files.
   - Note that the `_CS300_TEST_` value depends on if you are using a 64-bit or 32-bit system.
3. Compile your test application:
   ```
   $ gcc -std=c99 -D _GNU_SOURCE -static cs300_testapp.c -o cs300_testapp
   ```
     - `-D _GNU_SOURCE` allows access to the `syscall()` function defined in `unistd.h`.
     - `-static` causes all necessary library functions to be statically linked into the binary, thereby freeing us from having to ensure all required libraries are in the virtual machine.
   - This should produce an executable `cs300_testapp`.
     - Running this application on a normal kernel will have the syscall do nothing (return -1).
     ```
     $ ./cs300_testapp

     Diving to kernel level


     Rising to user level w/ result = -1
     ```
     - However, running it on the custom kernel (next section) will call your kernel code.

2.2 Running a Test Application

1. Boot your custom kernel using QEMU (as per the Custom Kernel Guide). Log in as `root`.
2. Transfer your `cs300_testapp` executable to your virtual machine (as per the Custom Kernel Guide). Command is likely:
   ```
   $ scp -P 2222 cs300_testapp root@localhost:~
   ```
   - Hint: Create a make file to build the program, and add a target to transfer to VM.
3. In your virtual machine, check that `cs300_testapp` has been transferred:
   ```
   # ls
   ```
   - You should see `cs300_testapp` in the listing.
4. In your virtual machine, run the test application. You should see the following:

```
root@debian-amd64:~# ./cs300_testapp
Diving to kernel level

[  186.746037] Hello World!
[  186.751607] --syscall argument 12345
[  186.751938] --returning 12345 + 1 = 12346

Rising to user level w/ result = 12346
root@debian-amd64:~#
```

   - If you see this, then congratulations! You have now written, complied, and finally called some kernel code! If not, see the troubleshooting section below.
5. Troubleshooting:
   - If your application returns the result -1 while running in your virtual machine, double check the following:
     - Your `cs300_testapp.c` file defines the system call number to match the number you entered in `syscall_64.tbl` (or `syscall_32.tlb` as the case may be).
     - Re-transfer your `cs300_testapp` executable to your virtual machine and rerun the command. (This may not be the problem, bit it's fast to try!)
     - Enure your custom kernel compiled correctly, and that you booted the correct kernel. You could check this by changing the "Local version" of the kernel (see Custom Kernel Guide), rebuild the kernel, and relaunch QEMU. Then check that your VM is running that new custom kernel. Display the kernel version using:
       ```
       # uname -a
       ```
     - Ensure you are running the correct version of the OS with the user-level code.
       - Both host OS and QEMU should have same architecture (number of bits):
         - On the host, check if it's 64 bit ("x86_64) or 32-bit ("i686") with:
           ```
           $ uname -m
           ```
         - Repeat the test on the target OS (QEMU):
           ```
           # uname -m
           ```
       - The user-level test code must be configured with the correct sys-call number matching the value you put in `syscall_64.tbl` (if a 64 bit system), or

`syscall_32.tbl` (if a 32-bit system).

- If the sys-call returns `12346` but you see nothing between "`Diving to kernel level`" and "`Rising to user level...`", then your VM may not be showing kernel messages (from `printk()`) by default. In that case you can run `dmesg` to see the output:

```
# dmesg
< ... omitted many lines of output...>
[  186.746037] Hello World!
[  186.751607] --syscall argument 12345
[  186.751938] --returning 12345 + 1 = 12346
```

- If your sys-calls always fails (return -1), double check the following:
  - You have downloaded your latest version of your test application.
  - You have correctly added the sys-call number to the `syscall_64.tbl` file.
  - Your sys-call numbers match in your application to `syscall_64.tbl`.
  - You have successfully recompiled your kernel code. To prove you are compiling the code, temporarily make an error in your kernel code implementing the sys-call and recompile the kernel. If the build fails on your code then you know it is being compiled; if not, you have a problem with your make files. After ensuring this, remove the error.

3. Further Resources

- Custom Kernel Guide for CMPT 300 by Brian Fraser.
- Guide to creating a new sys-call by Arrvindh Shriraman (2.xx kernel).
- Anatomy of a system call by David Drysdale.
- Linux kernel source code “Cross Reference” tool to find identifiers/text in the kernel, by Free Electrons.