Guide to Adding A Syscall to Linux 5.7+[1]

by Brian Fraser
Last update: June 9, 2020

This document guides the user through:

1. Adding a new system call (syscall) to Linux (as of 5.7.0)
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!
   ```
   (The # will be highlighted to draw attention to it being in QEMU!)
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.

Revision History

- June 3: Initial version
- June 8: Added introduction to each section, and hints on writing your own syscall.

1 Based on guide created by Arrvindh Shriraman.

1. Adding a HelloWorld System Call

Section Goal: A user-space application can communicate with the operating system via syscalls. We'll need to add a custom syscall for this assignment, so let's practice with a simple "hello world" system call!

This guide assumes you have the Linux source code (for version 5.7.0, or more recent may also work) in a folder `linux-5.7/`. **This guide focuses on 64-bit version of Linux**, but gives (untested) 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-5.7/` folder
   ```
   $ cd ~/cmpt300/linux-5.7
   ```
   - Adjust the path if necessary. See Custom Kernel Guide on course website for more.

3. Create a new folder named `cs300/` inside `linux-5.7/`
   ```
   $ mkdir cs300
   ```

4. Create a new file which implements your new system call (use your favourite editor):
   ```
   $ gedit cs300/cs300_test.c
   ```
   - Set file contents to:
     ```
     #include <linux/kernel.h>
     #include <linux/syscalls.h>

     // Implement a HelloWorld system call
     // Argument is passed from call in user space.

     SYSCALL_DEFINE1(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-5.7/`):
   ```
   $ cd ~/cmpt300/linux-5.7/
   $ gedit Makefile
   ```
   - Find the line which defines `core-y` (near line ~644)
     ```
     core-y            := usr/
     ```
   - Add your new folder to the end of the `core-y` define:
     ```
     core-y            := usr/ cs300/
     ```
   - Later when you make the kernel, it will also build the contents of your `cs300/` folder.
7. Actually create the new syscall by adding it to the kernel's list of system calls.

   On a 64 bit system:

   - Open the file which creates all the syscalls for the x86-64 bit architecture:
     ```
     $ cd ~/cmpt300/linux-5.7
     $ gedit ./arch/x86/entry/syscalls/syscall_64.tbl
     ```
   - Add the following lines at the end of the file:
     ```
     # CS300 Test SysCall
     548   common    cs300_test      sys_cs300_test
     ```
     - The number on the left is the syscall number you are creating. This syscall will exist only in your kernel (not world-wide Linux machines!). In general, once a syscall 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 syscalls (above it). If the previous last number was *N*, you should use *N+1*.
     - `common` defines the application binary interface.
     - `cs300_test` is name of the syscall, 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 syscall. This matches the name of the function created in `cs300_test.c`. (actually, the `SYSCALL_DEFINE1()` macro adds `sys_` to our function name)

   On a 32 bit system:
   - Open the file which creates all the syscalls for the x86-32 bit architecture:
     ```
     $ gedit arch/x86/entry/syscalls/syscall_32.tbl
     ```
   - Add the following line at the end of the file:

```
360   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 syscall! Now all you have to do is write some code which calls it (next section).
   ```
   $ make -j4
   ```

2. Calling your System Call

Section Goal: Write a user-space application to call our custom syscall because our custom OS kernel, which has our hello world syscall, needs a user-space application to call it in order for it to do anything. Note that syscalls are not accessible via the file system, so we need to write a user-space application to make the syscall.

2.1 Creating a Test Application

1. Outside of the `linux-5.7/` 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_ 548      // for a 64 bit system
//#define _CS300_TEST_ 360    // 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 syscall 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.

4. Troubleshootnig
   - If you get a linking error (such as below):

     Double check:
     - you filled in `syscall_64.tbl` correctly
     - your `cs300_test.c` function has the correct function name
     - your `syscall_64.tbl` and `cs300_testapp.c` use the same syscall number.

2.2 Running a Test Application

1. Boot your custom kernel using QEMU (as per the Custom Kernel Guide).
   Log in as `root`.
   Enable networking:
   `# ifup eth0`
2. Transfer your `cs300_testapp` executable to your virtual machine (as per the Custom Kernel Guide). Command is likely:
   `$ scp -P 10022 cs300_testapp root@localhost:~`
   - Hint: Create a make file to build the program, and add a target to transfer to VM.
3. In your QEMU 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 prints 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!)
- Ensure 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 syscall 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 syscall 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 syscalls always fails (return -1), double check the following:
  - You have downloaded your latest version of your test application.
  - You have correctly added the syscall number to the `syscall_64.tbl` file.
  - Your syscall 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 syscall 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. Tips on Writing a Syscall

- Each new syscall you wirte should have its own `.c` file in `linux-5.7/cs300/`
    - For example, if creating the syscall `awesome_call`, create the file: `~/cmpt300/linux-5.7/cs300/awesome_call.c`
    - Add `awesome_call.c` to the `linux-5.7/cs300/makefile` so that your syscall is compiled by the Linux kernel.
- Use `printk()` calls in the kernel to print out debug information. For the assignment, you *may* leave some of these `printk()` messages in your solution as these messages are not technically displayed by the user-space application. The messages you leave in should be helpful such as showing parameters values or errors it caught; they should not be of the sort "running line 17", or "past loop 1"...
    - Hint: `printk()` the parameters you are given, and `printk()` any error conditions you handle.
    - Hint: Add extra `printk()` messages to help debug where things are going wrong. Remember to remove these unneeded messages before submission.
- Correct memory access is one of the hardest part of writing a syscall which access a pointer or an array.
    - The kernel cannot trust anything it is given by a user-level application. Each pointer, for example, could be: a) perfectly fine, b) null, c) outside of the user program's memory space, or d) pointing to too small an area of memory. Since the kernel can read/write to any part of memory, it would be disastrous for the kernel to blindly trust a user-level pointer.
    - Each read you do using a pointer passed in as input (a user-level pointer) should be done via the `copy_from_user()` macro. This macro safely copies data from a user-level pointer to a kernel-level variable: if there's a problem reading or writing the memory, it stops and returns non-zero without crashing the kernel.
        - First use this macro to copy values into a local variables (which are in the kernel's memory space and thus safe to trust). Then, use these local variables in your program. See the Linux Kernel Development (ch5, p75) for more on the macro.
        - If working with an array, for example, create a local variable inside your syscall of the same type as the data in the user's array. Use `copy_from_user()` to copy one value at a time from the user's array into this local variable. If a copy fails (`copy_from_user()` returns non-zero) then have your syscall end immediately and likely return `-EFAULT`.

          Double check that you only ever access user pointers / arrays using `copy_from_user()`!
        - You *can* directly access non-pointer parameters to your syscall because they are passed by value so there is no possible problem access memory.
    - Likewise, when *writing to* a user-level pointer, use `copy_to_user()` which checks the pointer is valid (non-null), inside the user-program's memory space, and is writable (vs read-only).
        - Hint: Inside your syscall, create a local variable of type the same type as in the user-

space array. Compute the correct values in this local variable first, then at the very end use `copy_to_use()` to copy the contents to user's pointer / array.

- The kernel is compiled with C90; you must declare your variables at the start of a block (such as your function) vs in the middle of your function.

4. Further Resources

- 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.
- Another guide to creating a custom syscall.