

CSC 452 – Project 2: Syscalls

Due: Sunday, February 20, 2022, by 11:59pm

Project Description

We know from our description of the kernel that it provides fundamental operations that help us to interact with resources. These system calls include things like read, write, open, close, and fork. With access to the source code for the Linux kernel, we have the ability to extend the kernel's standard functionality with our own.

In this assignment we will be creating a simple interface for passing small text messages from one user to another by way of the kernel.

How it Will Work

There will be a userspace application called `osmsg` that allows a user to send and receive short text messages. To send a message the user will issue a command like:

```
osmsg -s jmisurda "Hello world"
```

which will queue up a message to be sent to the user `jmisurda` when they get their messages with the following command:

```
osmsg -r
```

This would output something like the following:

```
abc123 said: "Hello world"
```

Notice that this is more like email than instant messages, due to the fact that the recipient needs to explicitly request their messages. If there is more than one message waiting to be received, they will all be displayed at once. A message is discarded from the system after it has been read.

Syscall Interface

The `osmsg` application will do the sending and receiving of messages via two new system calls, which you will implement. The syscalls should look like the following:

```
asmlinkage long sys_csc452_send_msg(const char __user *to, const char __user
*msg, const char __user *from)
```

This function sends a single message given the three string parameters. The return value should be 0 for success, and a negative number on error.

```
asm linkage long sys_csc452_get_msg(const char __user *to, char __user *msg,
char __user *from)
```

This function takes the recipient's username as an input, and then returns via the two other parameters the message and sender. Notice that this only returns one message, but there may be many messages to deliver. The return value should be used to indicate if there are more messages to receive: 1 means to call the function again, 0 means that there are no more to deliver, and a negative number indicates an error. The msg and from pointers need to be pointing to space allocated in the user program's address space.

Implementation

There are two halves of implementation, the syscalls themselves, and the osmsg program.

For the syscalls, have them modify a linked list of messages. Memory in the kernel can be allocated by `kmalloc()`. You can make the nodes of the linked list structures with fixed sized fields. Make the fields reasonably-sized however.

Because this is a memory resident data structure, the messages will not survive a reboot. This is fine. You may want to make a shell script to add a lot of messages quickly into the VM.

Setting up the Kernel Source

Download the new Virtual Machine image from the website and follow these steps to configure:

1. Download the source code:

```
wget https://www.kernel.org/pub/linux/kernel/v5.x/linux-5.12.6.tar.xz
```

2. Extract

```
tar -xJf linux-5.12.6.tar.xz
```

3. Change into linux-5.12.6/ directory

```
cd linux-5.12.6
```

4. Make sure the kernel is clean and ready for our system

```
make clean && make mrproper
```

5. Use the current VM's device configuration to determine what drivers to build

```
make localmodconfig
```

And accept the defaults by hitting at the prompts

6. Configure the build to not use ZSTD compression

```
make menuconfig
```

- a. Hit on the selected "General setup ----->"
- b. Use the down arrow key to highlight "Kernel compression mode (ZSTD) ----->"
- c. Hit
- d. Use arrow keys to select "Bzip2". Hit .
- e. Use right arrow key to select "Save" (hit right twice)
- f. to select "< Ok >"
- g. Right arrow to select "Exit". .
- h. Right arrow to select "Exit". .
- i. You should now be back at the normal command line

7. Build (**this will take about 1 hour on a reasonably recent machine**)

```
make -j2
```

8. Build the drivers and install them

```
make modules_install
```

You should only need to do this once, however redoing step 4 (and on) will undo any changes you've made and give you a fresh copy of the kernel should things go horribly awry.

Rebuilding the Kernel

To build any changes you made, from the linux-5.12.6/ directory:

```
make -j2
```

```
make modules_install
```

Installing the Rebuilt Kernel

After you've built a kernel and done the `modules_install` step, you will need to install the kernel into the boot folder. From the `linux-5.12.6/` directory, do:

```
cp -v arch/x86/boot/bzImage /boot/vmlinuz-5.12.6

cp System.map /boot/System.map-5.12.6

mkinitcpio -k 5.12.6 -g /boot/initramfs-5.12.6.img
```

You also need to update the bootloader when the kernel changes. To do this (do it every time you install a new kernel if you like) type:

```
grub-mkconfig -o /boot/grub/grub.cfg
```

Booting into the Modified Kernel

As root, you simply can use the `reboot` command to cause the system to restart. When GRUB starts (the menu with the box around it) make sure to use the arrow keys to select the **Advanced Options for Arch Linux** option and hit enter. From there, select the **Arch Linux, with Linux 5.12.6** entry to boot into your new kernel. You will need to do this every time you want to use the kernel with your system calls.

This is deliberate, as if your kernel crashes, you can always boot into the “good” one to recover your system.

Adding a New Syscall

To add a new syscall to the Linux kernel, there are five main files that need to be modified:

1. `linux-5.12.6/kernel/sys.c`

This file contains the **actual implementation of the system calls**. We will declare them near the top to make editing easier. Use the `SYSCALL_DEFINEn()` macro to declare a system call with `n` parameters. For us, we will need three parameters. I suggest putting them **right above the `setpriority` system call** that already is there. Note that this **macro will automatically add the `sys_` prefix, so omit it when naming them here**.

2. `include/linux/syscalls.h`

This file needs to contain the **full prototypes of our two new system calls**, without using the `DEFINE` macro from `sys.c`.

3. `include/uapi/asm-generic/unistd.h`

This file contains the system call numbers (ordinals) for generic systems. We will add our two system calls following the other examples. We should **be adding system calls numbered 443 and 444**. Make sure

to also increase `__NR_syscalls` to 445 to indicate there are two more than there used to be. (NR should be read as “number”.)

4. `arch/x86/entry/syscalls/syscall_64.tbl`

This file declares the number that corresponds to the syscalls for 64-bit x86 systems. We will take the next two numbers:

```
443    common    csc452_send_msg    sys_csc452_send_msg
444    common    csc452_get_msg     sys_csc452_get_msg
```

5. `arch/x86/entry/syscalls/syscall_32.tbl`

This file declares the number that corresponds to the syscalls for 32-bit x86 systems. We again will take the next two numbers:

```
443    i386      csc452_send_msg    sys_csc452_send_msg
444    i386      csc452_get_msg     sys_csc452_get_msg
```

We now should rebuild the kernel, which since we touched a header file, will probably result in one more hour-long compilation. The good news is that we should only have to edit `sys.c` from now on, and that recompilation will be quick. After you build it, don't forget to follow the installation steps and reboot into the new kernel before you test.

Implementing and Building the `osmsg` Program

As you implement your syscalls, you are also going to want to test them via your co-developed `osmsg` program. The first thing we need is a way to use our new syscalls. We do this by using the `syscall()` function. The `syscall` function takes as its first parameter the number that represents which system call we would like to make. The remainder of the parameters are passed as the parameters to our `syscall` function. Since we are

We can write wrapper functions or macros to make the syscalls appear more natural in a C program. For example, since we are on the 32-bit version of x86, you could write:

```
int send_msg(char *to, char *msg, char *from) {
    syscall(443, to, msg, from);
}
```

And something similar for `get_msg()`.

Running `osmsg`

Make sure you run `osmsg` under your modified kernel. If you try to invoke a system call that isn't part of the kernel, it will appear to return -1.

File Backups

I suggest making a directory on Lectura under your home directory that no one else can see.

If you have not done so for the other projects, on Lectura, do:

```
mkdir private
```

```
chmod 700 private
```

Backup all the files you change under VirtualBox to your ~/private/ directory frequently!

Loss of work not backed up is not grounds for an extension. **YOU HAVE BEEN WARNED.**

Copying Files In and Out of VirtualBox

Once again, you can use scp (secure copy) to transfer files in and out of our virtual machine.

You can backup a file named sys.c to your private folder with:

```
scp sys.c USERNAME@lectura.cs.arizona.edu:private
```

Hints and Notes

- `printk()` is the version of `printf()` you can use for debugging messages from the kernel.
 - Unless you configure the `/proc/sys/kernel/printk` file to contain values of at least 5, you won't see the message printed on the screen and will have to use the output of the `dmesg` command to see all kernel messages.
- In general, you can use some library standard C functions, but not all. If they do an OS call, they may not work
- `kmalloc()` and `kfree()` allow you to allocate kernel memory to make nodes for a linked-list based queue. Use `GFP_KERNEL` for the mode.
- To get the current user's name, use `getenv()` to get the appropriate environment variable. Note this is in no way a secure authentication method
- Use the linux command `useradd` to add more users to your system. You can use `su` to switch users and `passwd` (as root) to change the password of the users. Use the `man` pages or google to learn more about these commands.

- This is not an appropriate system call for a real operating system since it does not provide anything a user level program could not do itself. It is for illustrative purposes only.

Requirements and Submission

You need to submit:

- Your well-commented osmsg program's source
- sys.c containing your implementation of the system calls

We will use the turnin program on lectura to turn in your project. If your files are named sys.c and osmsg.c, execute the following command from lectura in the directory that contains them:

```
turnin csc452-spring22-p2 sys.c osmsg.c
```