OS support on IMS report #1

Yusen Liu
liu797@wisc.edu

Jan. 8 ~Jan. 12

Abstract

In order to dig into how a linux kernel works, I picked Ubuntu Desktop 20.04.1 with original kernel verison 5.8.0-36-generic and installed it on VMware. I followed the online materials and compiled the linux kernel of version 5.8.1. After compilation succeeded, I added several system calls and tested their functionalities. I wrote a script that automated kernel compilation.

1 Introduction

Linux is a widely used operating system that is open source. In order to get some sense of how Linux works, we need to first obtain a Linux environment. I searched on the internet and compared the differences between Ubuntu and other operating systems such as CentOS. Since Ubuntu has a lot more community support, and it is frequently updated[1], I thought Ubuntu would be the best choice. I installed Ubuntu Desktop 20.04.1 LTS on VMware because it is recommended to work with the latest code[2].

Since the release of new hardware devices are frequent, in order to make those new hardware devices work on Ubuntu, Hardware Enablement (HWE) was introduced[3]. Basically, the first solution is to use rolling releases for the kernel: as soon as a new kernel is released, it is packaged for Ubuntu, tested (via the proposed pocket and special Q/A methodologies), and made available to Ubuntu users[4]. The disadvantages of this solution are: some bugs or issues are introduced because of releasing a new kernel too quickly, and thus not suitable for the enterprise. The second solution is to offer users different kernels. Therefore Ubuntu will offer at least two kernels: the General Availability (GA) kernel, which is the most stable kernel that does not get updated to point releases; and the Hardware Enablement (HWE) kernel, which is the most recent kernel released.

System calls provide a layer between the hardware and user space processes[2]. In other words, system calls provide an abstracted hardware interface for user space. In Linux, system calls are the only means user space has to interface with the kernel that provide

a return value of type `long` that signifies success or error (usually zero on success, and negative return value on failure). When a system call returns an error, the C library would write an error code into `errno`, which is a global variable that can be translated into human-readable errors using `perror()`. Taking the system call `getpid()` as an example:

```
SYSCALL_DEFINE0(getpid)
{
  return task_tgid_vnr(current); // current->tgid
}

/* The expanded code is below */
asmlinkage long sys_getpid(void);
```

Note that[2]:

- `SYSCALL_DEFINE0` is a macro that defines a system call with 0 parameters. Hence, `SYSCALL_DEFINEn` means defining a system call with n parameters. One thing to mention is that, the number of parameters of `SYSCALL_DEFINEn` is $(2n + 1)$ where the first parameter is the name of the system call.
- `asmlinkage` is a directive to tell the compiler to look only on the stack for the function's arguments, which is a required modifier for all system calls.
- `sys_getpid` is the naming convention of all system calls in Linux.
- Each system call is assigned a unique system call number that referred by the process. The kernel keeps a list of all registered system calls in the system call table. Under my implementation, it is stored in `~/kernel/linux-5.8.1/arch/x86/entry/syscalls/syscall_64.tbl`. See Section 2.2.1 for more details.

2 Methodology and Results

2.1 Compiling the kernel

2.1.1 Original kernel

Right after I setup the Linux environment, which is Ubuntu 20.04.1 LTS Desktop, I ran a command `$ uname -r` to check the original version of the kernel that was running, after which, I got 5.8.0-36-generic.

Next, I checked the `/lib/modules` directory and there were two directories inside: 5.4.0-42-generic and 5.8.0-36-generic. The `/lib/modules/$(uname -r)` directory stores all compiled drivers under Linux operating system[5], i.e. files that contain the code of a kernel module. The files in that directory contain the modules that we can load into

the running kernel[6]. In my environment, it was 5.8.0-36-generic. One can use the `$ modprobe` command to intelligently add or remove a module from the Linux kernel.

2.1.2 Kernel compilation

I downloaded the kernel tarball of version 5.8.1 from kernel.org and unzipped it into `~/kernel` directory. Note that, the kernel source is typically installed in `/usr/src/linux` directory. It is not recommended to use that source tree for development because the kernel version against with C library is compiled is often linked to the tree[2]. Hence, when installing the new kernel or performing some experiments on the new kernel, `/usr/src/linux` should remain untouched. On Ubuntu 20.04.1 LTS Desktop, there are four directories inside `/usr/src`, which are linux-headers-5.4.0-42, linux-headers-5.4.0-42-generic, linux-headers-5.8.0-36-generic and linux-hwe-5.8-headers-5.8.0-36. The last one corresponds to the HWE kernel.

In order to compile the kernel, we need to change the current working directory to the `~/kernel/linux-5.8.1` directory and execute several commands, which are listed below[2][7][8].

1. Fully update the operating system and install the requirements for compiling the new kernel. Note that this step could also be done right after we have setup the environment.
   - `$ sudo apt update && sudo apt upgrade -y`
   - `$ sudo apt install build-essential libncurses-dev libssl-dev -y`
   - `$ sudo apt install libelf-dev bison flex emacs -y`
   - `$ sudo apt clean && sudo apt autoremove -y`
2. Configure the new kernel for only the specific features and drivers we want. There are many ways to configure the kernel: we can simply copy the configuration from the current kernel's .config file, or we could create a configuration based on the defaults for the architecture, or we could use an ncurses-based graphical utility to configure.
   - `$ cp /boot/config-$(uname -r) .config`
   - or `$ make defconfig`
   - or `$ make menuconfig`
3. Validate and update the configuration.
   - `$ make oldconfig`
4. Compile the new kernel's source code, i.e. build the kernel after the kernel configuration is set. Note that this would take a huge amount of time (2 hours 45 minutes on my computer), and make sure to have at least 12GB of free space available.

- `$ make`

5. Prepare the installer of the new kernel, i.e. install the modules we have enabled to their correct home under `/lib/modules`.
   - `$ sudo make modules_install`
6. Install the new kernel.
   - `$ sudo make install`
7. Update the bootloader of the operating system with the new kernel after we have installed the new kernel.
   - `$ sudo update-initramfs -c -k 5.8.1`
   - `$ sudo update-grub`

After successfully executing the above commands step by step, need to reboot the operating system and use command `$ uname -r` to check for the new kernel. In my experiment, the result was 5.8.1, which meant the kernel had been successfully compiled and installed. Inside the `/lib/modules` directory, we could see that there was a new directory called 5.8.1.

2.2 Adding a system call

After compilation of the new kernel, I continued by adding system calls to investigate how system calls were implemented.

2.2.1 my_first_syscall()

My first approach was following the online materials[7]. The steps are listed below.

1. Change the working directory to the root directory of the recently unpacked source code (5.11.0-rc2 in this case).
   - `$ cd ~/kernel/linux-5.8.1`
2. Create a home directory of my own system call. I picked my_syscalls as the name of the directory.
   - `$ mkdir my_syscalls`
3. Create a C file for the new system call inside `./my_syscalls`.

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

  SYSCALL_DEFINE0(my_first_syscall)

  {
      printk("This is my first system call.\n");
      return 0;
  }
  ```

4. Create a Makefile for the system call inside `./my_syscalls` directory.

   - `obj-y := my_first_syscall.o`

5. Add the home directory of the new system call to the main Makefile of the kernel. Make sure the current working directory is `~/kernel/linux-5.8.1`. Edit Makefile by firstly searching for the second "core-y", where there is a list of directories: `kernel/ certs/ mm/ fs/ ipc/ security/ crypto/ block/`, and then append my_syscalls to the end of this line: `kernel/ certs/ mm/ fs/ ipc/ security/ crypto/ block/ my_syscalls/`

6. Add the corresponding function prototype for the system call to the header file of system calls, of which the path is `./include/linux/syscalls.h`. Add the following code to the end of syscalls.h before `#endif`:

   - `asmlinkage long sys_my_first_syscall(void);`

7. Add the system call to the kernel's system call table, whose path is `./arch/x86/entry/syscalls/syscall_64.tbl`. After opening the file syscall_64.tbl, navigate to line 362, which is the system call `faccessat2` with number 439. We add our own system call - `my_first_syscall` after line 362, with number 440, which is shown below:

   - ```
     ...
     439   common    faccessat2              sys_faccessat2
     440   common    my_first_syscall         sys_my_first_syscall
     #
     ...
     ```

What we need to do next is to compile the kernel, which is discussed in Section2.1.2

2.2.2 Functionality test of my_first_syscall()

After recompiling the kernel, we need to test the functionality of `my_first_syscall()`. I created a C file, which is shown below:

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

#define __NR_my_first_syscall 440

int main(int argc, char *argv[])
{
    long ret_val = syscall(__NR_my_first_syscall);

    if(ret_val < 0){
        perror("syscall failed\n");
    }
    else{
        printf("syscall succeed\n");
    }

    return 0;
}
```

The output was: `syscall succeed`. After executing the command `$ dmesg` to check the kernel log buffer, we could see `[ 98.744208] This is my first syscall.` on the last line, which means that `printk()` inside `my_first_syscall()` successfully prints to the kernel log buffer.

2.2.3 Another system call - wcpy_from_user()

The next system call is `wcpy_from_user()`[9] where w represents Wisconsin. Prior to dive into details of implementation, we need to introduce a kernel API: `strncpy_from_user()` that used for copying a null-terminated string from user space to kernel space. On success, the function returns the length of the string. If access to user space fails, it will return `-EFAULT`, i.e. return a negative value. The code is as follows:

```
/*
   The directory of this file is
   ~/kernel/kernel-5.8.1/my_syscall/wcpy_from_user.c
*/

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

```
SYSCALL_DEFINE1(wcpy_from_user,
char*, src)
{
  char buf[128];
  long cpy_result = strncpy_from_user(buf, src, sizeof(buf));

  /* if failure on accessing user space,
     or size of buf is less than length of src */
  if(cpy_result < 0 || cpy_result == sizeof(buf))
    return -EFAULT;

  printk(KERN_INFO "copied from user: %s\n", buf);
  return 0;
}
```

Note that:

- Since this is the second system call, we do not need to create another directory. Instead, we could simply create `wcpy_from_user.c` in directory `my_syscall`.
- When modifying `Makefile` inside the directory `my_syscall`, simply append to the end of the first line: `obj-y := my_first_syscall.o wcpy_from_user.o`.

2.2.4 Functionality test of wcpy_from_user()

After recompiling the kernel, we need to test the functionality of `wcpy_from_user()`. The testing code is as follows:

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

#define SYS_wcpy_from_user 441

int main(int argc, char *argv[])
{
  if(argc <= 1){
    printf("A string is required to test the syscall - wcpy_from_user().\n");
  }

  char* src = argv[1];
  printf("user input is: %s\n", src);
```

```
  long ret_val = syscall(SYS_wcpy_from_user, src);

    if(ret_val < 0){
        perror("syscall failed.\n");
    }
    else{
        printf("syscall succeed.\n");
    }
    return 0;
}
```

The output was: syscall failed. After executing the command `$ dmesg` to check the kernel log buffer, I noticed that there was one line saying that input handler disabled. However, after running the testing code for several times, the kernel log buffer did not output anything new. I ran the `fuser /dev/rfkill` and got 1288. `ps -p 1288 -o comm=` would give me `gsd-rfkill`. According to the Fedora Documentation[10], this has nothing to do with the failure of the system call `wcpy_from_user()`. Hence, I am still working on figuring out why it happened.

3 Discussion

As discussed in Section 2.1.2, compiling the kernel would take a huge amount of time. My hardware setup of the virtual machine is 4 GB of memory and 2 cores. I tried to use the '-j' option but my virtual machine died because of that. I thought it was because of insufficient memory or disk size at the beginning. Hence, I deleted the virtual machine and reinstall Ubuntu for a lot of times trying to find the optimal disk size. However, even with the 8 GB of memory and 4 cores with 200 GB disk size setup, '-j' option would still lead to the death of my virtual machine. Note that my hardware is 16 GB with 6 cores and 500 GB available disk size. The only solution to successfully compiling the kernel is to discard the '-j' option, but it is less productive to add a system call since compiling the kernel takes up so much time. However, there is an option - compiling the mainline kernel, and it would take much less time (20 minutes). The main reason is the mainline kernels do not include any Ubuntu-provided drivers or patches[11]. Since it is not recommended to work with mainline kernels, I would keep to the kernel verison 5.8.1. Another way of saving time is to write a shell script to automate kernel compilation, which discussed in Section 5.

The next thing I would like to discuss about is that, we cannot rely on online tutorials only. I followed one tutorial at first without thinking too much when I was trying to compile the kernel[7]. However, when rebooting the system, there was an alert saying a UUID did not exist. I started over and recompile the kernel for two times but still failed. Hence I searched for other tutorials and found one critical line of code:

`sudo update-initramfs -c -k 5.8.1` which generates an initramfs image that the first tutorial did not cover.

This report can be also served as a reference for me when compiling the kernel and adding some system calls.

4 Future plan

I purchased the book *Linux Kernel Development* at the beginning of year 2021, and found a lot of things useful from that book. What I am planning to do next is to go on reading the book. Since I realize that there is still a lot of system calls that I am not familiar with, I would simultaneously go through the Linux man-pages (https://www.kernel.org/doc/man-pages/). I also found an interesting website that includes source code of GNU C library (https://code.woboq.org/userspace/glibc/), which would help me better understand the implementation of functions that I used to use in CS 537. Another reference that is useful is the Linux source tree (https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/).

5 Appendix

Enlighted by online resourses[9], I could write a script that automatically compiles the kernel and reboot the system after installing the kernel. The script is shown below:

```
#!/usr/bin/bash
# Compile and deploy a kernel

make
make modules_install
make install
update-initramfs -c -k 5.8.1
update-grub
reboot
```

Note that when running the script, need to run `$ sudo deploy.sh` so that all commands within the script will be run with root privileges, and it is only required to give the password when launching the script.

References

[1] Domantas G. Centos vs ubuntu – which one to choose for your web server, 2019.

[2] Robert Love. *Linux kernel development*. Pearson Education, 2010.

[3] guiverc. Rollingltsenablementstack, 2021.

[4] Andrea Corbellini. What is hardware enablement (hwe), 2013.

[5] Vivek Gite. Find out linux kernel modules (drivers) location / directory, 2017.

[6] Gilles. Difference between /lib/module/$(uname -r) and /sys/module, 2015.

[7] Jihan Jasper Al-rashid. Adding a system call to the linux kernel (5.8.1) in ubuntu (20.04 lts), 2020.

[8] Jack Wallen. How to compile a linux kernel, 2018.

[9] Stephen Brennan. Tutorial - write a system call, 2016.

[10] Rüdiger Landmann Don Domingo.

[11] ish. Should i upgrade to the “mainline” kernels, 2020.