CS 140 Lab Exercise 10: Modifying the Linux Kernel

Department of Computer Science
College of Engineering
University of the Philippines Diliman

1 Overview

This laboratory involves modifying the Linux 6.0.9 kernel.

At the end of this activity, you should have learned:

- How Linux represents the x86-64 syscall table
- How to make direct modifications to the Linux kernel via kernel recompilation

Note: You are strongly encouraged to **start as early as possible**—Linux kernel recompilation may take more than an hour using a single CPU core.

2 Prerequisites

2.1 Background

x86-64 is the 64-bit variant of the 32-bit x86 architecture and is the architecture used by most modern computers. While there is no need to know x86-64 assembly for this exercise, you should take note that x86-64 shares a lot of common code with x86 in the Linux kernel.

The Linux kernel itself is coded in C and may be extended either by recompilation or by plugging in Linux Kernel Modules (LKMs)—only the former will be covered in this exercise.

2.2 Installation

For this exercise, you will need an x86-64 machine[1] running Linux (preferably via a virtual machine). Ensure that prerequisites for kernel compilation are installed:

- `sudo apt install -y build-essential gcc make perl libncurses-dev flex bison libssl-dev libelf-dev`

If you are running the said Linux instance on VirtualBox, you are strongly encouraged to set the number of available processors to the maximum allowable value via **Settings** > **System** > **Processor** for faster compilation times.

3 Walkthrough: Linux Kernel Recompilation

In this section, we will be going through how the `hello` and `getname` syscalls from Lab Exercise 1 can be implemented in Linux via editing kernel source code.

[1]Most modern computers excluding those using Apple Silicon (M1/M2)

3.1 Linux kernel compilation

3.1.1 Kernel source code

To compile the Linux kernel, you must have its source code. While you may find this via official repositories[2], you are required to use the Linux 6.0.9 kernel source code in your Github Classroom repository:

`https://classroom.github.com/a/V6h6zz7M`

Before moving forward, ensure that your Linux instance has access to the source code. For those using VirtualBox, you may opt to set up a shared folder[3] containing the source code—this allows you easily edit the code from your host machine.

3.1.2 Setting up `.config`

Linux kernel compilation depends on a config file called `.config` containing machine-specific settings. To generate this, open a terminal instance inside the root directory of your Linux kernel source code *(assume this to be the working directory for all commands mentioned below)* and run `yes '' | make localmodconfig`—this sets all configuration values to their defaults.

Next, open the generated `.config` using a text editor and make the following change to the relevant line:

- Before: `CONFIG_SYSTEM_TRUSTED_KEYS="debian/canonical-revoked-certs.pem"`
- After: `CONFIG_SYSTEM_TRUSTED_KEYS=""`

3.1.3 Compilation

To compile the Linux kernel using all available CPU cores, run ``make -j`nproc` ``. Note that the command uses backticks (key to the left of `1` on a standard QWERTY keyboard), not quotes. `nproc` is a utility that returns the number of CPU cores available to Linux. The backticks paste the output of the surrounded (inner) command to the outer command being executed while the `-j` flag sets how many cores the compilation process will utilize. If `nproc` outputs `4`, ``make -j`nproc` `` is equivalent to `make -j4`.

Note that compiling the Linux kernel may take around 30 minutes for a four-core setup. To verify that the compilation process succeeded, the following files must be present in the root directory:

1. `vmlinux`
2. `vmlinux.map`
3. `System.map`

3.1.4 Installation

Once the kernel output files are compiled, execute the following commands:

1. `sudo make modules_install` *(installs all kernel modules)*

[2] `https://mirrors.edge.kernel.org/pub/linux/kernel/`
[3] See Lab Exercise 0 for instructions on how to set this up

2. `sudo make install` *(installs the kernel itself into the hard disk)*

The new kernel should now be accessible via the GRUB bootloader screen. Reboot your Linux instance, hold down the `Shift` key during startup to open GRUB, select **Advanced options for Ubuntu**, and choose the new 6.0.9 kernel. Note that pressing the `Escape` key while the Ubuntu logo is on screen shows the startup progress of Linux.

Once Ubuntu has loaded, open a terminal instance and run `uname -mrs`—this should display `Linux 6.0.9 x86_64` as its output.

3.2 The `hello` syscall

Recall that the `hello` syscall is a zero-parameter syscall that makes the kernel print a message. In Linux, `printk` outputs to the *kernel log buffer* instead of standard output.

3.2.1 Making a new system call: Kernel side

To define a new system call in Linux, the following must be done:

1. Add a new entry in the syscall table of each target architecture
2. Define its syscall handler (ideally in a new file)

To add the `hello` syscall to the x86-64 syscall table of Linux, add the following line with at least one space or tab between tokens to the bottom of `arch/x86/entry/syscalls/syscall_64.tbl`:

```
548 64 hello sys_hello
```

The above defines the `hello` syscall as syscall code `548` with handler `sys_hello` for just x86-64 (`64`). To define the implementation of `sys_hello`, do the following:

1. Create a directory called `lab10` under the root directory of the kernel source code
2. Create a file called `hello.c` in the `lab10` directory with content as defined in Code Block 1
3. Create a file called `Makefile` in the `lab10` directory with content `obj-y := hello.o`
4. Add `lab10/` as the last token of `core-y` in `Makefile` (after `arch/$(SRCARCH)/` and separated by at least once space)
5. Add `long sys_hello(void);` just before the last `#endif` at the bottom of `include/linux/syscalls.h` *(makes `sys_hello` accessible in other parts of the kernel)*

Note that modifying the above files will cause `make` to recompile the entire kernel. To avoid wastage of time, double-check that your changes are correct.

Similar to xv6, syscall handlers in Linux are expected to return `0` to indicate success. Linux requires syscall handlers to have a return type of `long`.

In Linux, special syscall macros such as `SYSCALL_DEFINE0`, `SYSCALL_DEFINE1`, and `SYSCALL_DEFINE1` (up to `6`) must be used when defining syscall handlers. The integer suffix of each macro determines the number of arguments the handler is expecting. The said macros must be supplied the following, separated by commas:

- Syscall name (as defined in the syscall table)
- For each argument:
  - Argument data type
  - Argument name

Recompile the Linux kernel via `make -j`nproc``, install it via `sudo make modules_install` and `sudo make install`, and reboot the system with the new kernel.

Code Block 1 Syscall handler implementation for `hello`

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

SYSCALL_DEFINE0(hello) {
    printk("Hello from the Linux kernel!\n");
    return 0;
}
```

3.2.2 Making a new system call: User side

To test that your `hello` syscall works, make a user program with code in Code Block 2, compile it via `gcc`, and run it. To see the output of `printk`, run `sudo dmesg`.

Code Block 2 User test program for the `hello` syscall (defined as syscall code 548)

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

int main() {
    syscall(548);
}
```

Note that in contrast to xv6, user programs in Linux are **not** compiled alongside the kernel.

3.3 The `getname` syscall

Recall that the `getname` syscall is a single-parameter syscall that takes in a `char *`, stores the name of the invoking process into the `char *`, and returns the length of the said name.

To implement this, do the following:

1. Add `549 64 getname sys_getname` to `arch/x86/entry/syscalls/syscall_64.tbl`
2. Create a file `lab10/getname.c` with content as defined in Code Block 3
3. Change `obj-y := hello.o` to `obj-y := hello.o getname.o` in `lab10/Makefile`

4. Add `int sys_getname(char *);` to `include/linux/syscalls.h`
5. Recompile and reinstall the kernel
6. Compile and run the code in Code Block 4 as a user program and verify that its output is correct

Code Block 3 Syscall handler implementation for `getname`

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

SYSCALL_DEFINE1(getname, char *, buffer) {
    printk("getname: %s\n", get_current()->comm);
    strcpy(buffer, get_current()->comm);

    return strlen(buffer);
}
```

Code Block 4 User test program for the `getname` syscall (defined as syscall code `549`)

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

int main() {
    char buffer[100];
    long length = syscall(549, buffer);

    printf("Process name is: %s\n", buffer);
    printf("Name is %ld characters long\n", length);
}
```

In Linux, `get_current()` returns a per-CPU global pointer variable that refers to the PCB (with type `struct task_struct`) of the currently active process. `struct task_struct` is defined in `include/linux/sched.h`[4]. The `comm` field of a `struct task_struct` stores the executable name of the associated process.

4 Lab Exercise

4.1 `setname`

Create a new system call `setname` with syscall number `550` that replaces the characters in the `get_current()->comm` field of the invoking process with the characters of the string argument.

[4]You may use the *Elixir Cross Referencer* to more easily go through Linux kernel code: `https://elixir.bootlin.com/linux/v6.0.9/source/include/linux/sched.h#L727`

While you are not expected to create a user-facing interface for `setname` (i.e., `syscall(550, ...);` will suffice for testing), it is expected to look like this:

```
long setname(char *)
```

If the string argument has more than 15 characters before the null terminator (i.e., the null terminator is beyond index `15`) or is empty (i.e., index `0` contains `'\0'`), `setname` must **not** change `get_current()->comm` and must return `-1` (`0` otherwise).

4.2 `disable`

Before starting on this item, commit what you have so far.

Create a new system call `disable` with syscall number `551` that takes in a single `int` argument corresponding to a syscall number and removes its associated syscall handler entry from the syscall table.

Invoking a removed syscall should yield the same error message as that of a call to a nonexistent syscall (see `do_syscall_64` in `arch/x86/entry/common.c`[5]).

`disable` should return `-1` if the syscall number argument refers to a nonexistent or already disabled syscall, or `0` otherwise. `disable` is allowed to disable syscall number `551` (i.e., itself).

While you are not expected to create a user-facing interface for `disable` (i.e., `syscall(551, ...);` will suffice for testing), it is expected to look like this:

```
long disable(int n)
```

5 Lab Report

Submit your laboratory report as `cs140lab10.pdf` through the UVLê submission module. You may submit multiple times before the specified deadline. **This is to be done individually.**

Deadline: 11 December 2022, 11:59 PM (before Monday)
Maximum score: 100/100

1. *[50pts]* Explain how your `setname` syscall implementation works. Ensure that the code is properly pushed to your Github Classroom repository.
2. *[50pts]* Explain how your `disable` syscall implementation works. Ensure that the code is properly pushed to your Github Classroom repository.

[5] `https://elixir.bootlin.com/linux/v6.0.9/source/arch/x86/entry/common.c#L73`