The University of Texas at Arlington

CSE 3320 - Spring 2020
Operating Systems
Project 1 - Process and Thread

Instructor: Jia Rao

Points Possible: 100
Handed out: Jan. 31, 2020
Due date: 11:59 pm, Saturday, Feb. 15, 2020

Introduction

The purpose of this project is to study how processes and threads are managed in a *Linux* environment. The objectives of this project is to learn:

1. Get familiar with the *Linux* environment.
2. How to build a *Linux* kernel.
3. How to add a new system call in the *Linux* kernel.
4. How to obtain various information for a running process/thread from the *Linux* kernel.

Project submission

For each project, create a gzipped file containing the following items, and submit it through Blackboard.

1. A report that briefly describes how you solved the problems and what you have learned. For example, you may want to make a *typescript* of the steps in building the *Linux* kernel; or the changes you made in the *Linux* kernel source for adding a new system call.
2. The programming codes including both the kernel-level codes you added and the user-level testing program.

Assignments

Assignment 0: Build the *Linux* kernel (20 points)

Step 1: Get the *Linux* kernel code
Before you download and compile the *Linux* kernel source, make sure you have development tools installed

on your system.
In CentOS, install these software using `yum` [1]:

```
# yum install -y gcc ncurses-devel make wget
```

In Ubuntu, install these software using `apt`:

```
# apt install -y gcc libncurses5-dev make wget flex bison vim libssl-dev libelf-dev
```

Visit `http://kernel.org` and download the source code of your current running kernel. To obtain the version of your current kernel, type:

```
$ uname -r
```

```
$ 2.6.32-220.el6.i686
```

The suffix `i686` indicates that the kernel is 32 bit. A 64-bit kernel will show a suffix of `x86_64`.

Then, download kernel `2.6.32` and extract the source:

```
$ wget http://www.kernel.org/pub/linux/kernel/v2.6/linux-2.6.32.tar.gz
```

```
$ tar xvzf linux-2.6.32.tar.gz
```

Note that you may not use the above link for kernel download as your kernel may be different (much newer) than the kernel shown here. We will refer `LINUX_SOURCE` to the top directory of the kernel source.

Step 2: Configure your new kernel
Before compiling the new kernel, a `.config` file needs to be generated in the top directory of the kernel source. To generate the config file and make possible changes to the default kernel configurations, type:

```
$ make menuconfig
```

No changes to the default configuration are needed at this time. Press `ESC` to exit the configuration menu and a default config file will be generated.
Step 3: Compile the kernel
In `LINUX_SOURCE`, compile to create a compressed kernel image:

```
$ make -j4
```

Option `-j4` will utilize four CPUs to accelerate the compilation.

To compile kernel modules:

```
$ make modules
```

Step 4: Install the kernel
Install kernel modules (become a root user, use the `su` command):

```
$ su -
```

```
# make modules_install
```

Install the kernel:

```
# make install
```

If you are using Ubuntu, you need to create an init ramdisk manually:

```
$ sudo mkinitramfs -o /boot/initrd.img-2.6.32
```

```
$ sudo update-initramfs -c -k 2.6.32
```

The kernel image and other related files have been installed into the `/boot` directory.
Step 5: Modify grub configuration file
1. If you are using CentOS:
Change the grub configuration file to boot from the newly installed kernel. Open file using `vim`:

[1] The `#` prompt indicates that this command requires root privilege. `$` indicates user privilege.

```
# vim /boot/grub/menu.lst
```

The newly installed kernel should have a booting order 0, change the default kernel to 0:

```
default=0
```

IMPORTANT: In case that the new kernel fails to boot, we may want to select the old kernel from the VM console. Set the `timeout` value (in seconds) to a large value (e.g., 25) to give yourself enough time to select.

```
timeout=25
```

2. If you are using Ubuntu:
Change the grub configuration file:

```
$ sudo vim /etc/default/grub
```

Make the following changes:

```
GRUB_DEFAULT=2
GRUB_TIMEOUT_STYLE=menu
GRUB_TIMEOUT=25
```

Then, update the grub entry:

```
$ sudo update-grub2
```

Step 6: Reboot your VM
Reboot to the new kernel:

```
# reboot
```

After boot, check if you have the new kernel:

```
$ uname -r
$ 2.6.32
```

Assignment 1: Add a new system call into the *Linux* kernel (30 points)

In this assignment, we add a simple system call `helloworld` to the *Linux* kernel. The system call prints out a hello world message to the syslog. You need to implement the system call in the kernel and write a user-level program to test your new system call.

Step 1: Implement your system call Change your current working directory to the kernel source directory.

```
$ cd LINUX_SOURCE
```

Make a new directory `my_source`to contain your implementation:

```
$ mkdir my_source
```

Create a C file and implement your system call here:

```
$ touch my_source/sys_helloworld.c
```

Edit the source code to include the following implementation:

```
$ vim my_source/sys_helloworld.c
```

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

SYSCALL_DEFINE0(helloworld)
{
  printk(KERN_EMERG "Hello World !\n");
  return 1;
```

```
}
```

Add a Makefile to the `my_source` folder:

```
$ touch my_source/Makefile
```

```
$ vim my_source/Makefile
```

```
#
#Makefile of the new system call
#
obj-y := sys_helloworld.o
```

Modify the Makefile in the top directory to include the new system call in the kernel compilation:

```
$ vim Makefile
```

Find the line where `core-y` is defined and add the `my_source` directory to it:

```
core-y += kernel/ mm/ fs/ ipc/ security/ crypto/ block/ my_source/
```

32-bit Linux Add a function pointer to the new system call in `arch/x86/kernel/syscall_table_32.S`:

```
$ vim arch/x86/kernel/syscall_table_32.S
```

Add the following line to the end of this file:

```
.long sys_helloworld /* 337 */
```

Now you have defined `helloworld` as system call 337.

Register your system call with the kernel by editing `arch/x86/include/asm/unistd_32.h`. The last system call in `arch/x86/kernel/syscall_table_32.S` has an id number 336, then our new system call should be numbered as 337. Add this to the file `unistd_32.h`, right below the definition of system call 336:

```
#define __NR_helloworld 337
```

Modify the total number of system calls to $337 + 1 = 338$:

```
#define NR_syscalls 338
```

Declare the system call routine. Add the following to the end of arch/x86/include/asm/syscall.h (right before the line `CONFIG_X86_32`):

```
asmlinkage int sys_helloworld(void);
```

64-bit Linux Add the new system call to the system call table. Open the `arch/x86/entry/syscalls/syscall_64.tbl` file, and go to the part of the file right before the beginning of the "x32-specific system call numbers" , and add the following after the `rseq` line:

```
$ 335 common helloworld __x64_sys_helloworld
```

Declare your new syscall in the header file `include/linux/syscalls.h`. Go to the bottom of the file and add the following right before the `#endif`:

```
$ asmlinkage long sys_helloworld(void);
```

Repeat step 3 and 4 in assignment 0 to re-compile the kernel and reboot to the new kernel.

Step 2: Write a user-level program to test your system call

Go to your home directory and create a test program `test_syscall.c`:

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

```
#define __NR_helloworld 335

int main(int argc, char *argv[])
{
  syscall(__NR_helloworld);
  return 0;
}
```

The syscall number should match the syscall ID you used in the kernel.
Compile the program:

```
$ gcc test_syscall.c -o test_syscall
```

Test the new system call by running:

```
$ ./test_syscall
```

The test program will call the new system call and output a helloworld message at the tail of the output of `dmesg`.

Assignment 2: Extend your new system call to print out the calling process's information (35 points)

Follow the instructions we discussed above and implement another system call `print_self`. This system call identifies the calling process at the user-level and print out various information of the process.

Implement the `print_self` system call and print out the following information of the calling process:

- Process id, running state, and program name
- Start time and virtual runtime
- Its parent processes until `init`

HINT: The macro `current` returns a pointer to the `task_struct` of the current running process. See the following link for more information:
`http://linuxgazette.net/133/saha.html`

Assignment 3: Extend your new system call to print out the information of an arbitrary process identified by its `PID`(15 points)

Implement another system call `print_other` to print the information for an arbitrary process. The system call takes a process pid as its argument and outputs the above information of this process.

HINT: You can start from the `init` process and iterate over all the processes. For each process, compare its `pid` with the target pid. If there is a match, return the pointer to this `task_struct`.
A better approach is to use the `pidhash` table to look up the process in the process table. `Linux` provides many functions to find a task by its pid.

Tips:

- The recent Linux requires much storage space after compilation. It is recommended that you at least configure 35GB disk space for your VM.
- Configure 2-4 CPUs for your VM so as to use multiple CPUs for compilation. The first time compilation of the Linux kernel takes 1-2 hours.
- Search "how to configure port forwarding" in VirtualBox to enable `ssh` remote login on your VM. The remote login is much faster than using the console interface.