University of Colorado at Colorado Springs
CS4500/5500 - Fall 2016 Operating Systems
Project 1 - Processes and Threads

Instructor: Yanyan Zhuang
Total Points: 100
Handed out: Sept. 2, 2016
Due date: 11:59 pm, Monday, Sept. 19, 2016

Introduction

The purpose of this project is to study how processes and threads are managed by the Linux OS. The objectives of this project is to:

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

Project submission

For each project, please create a zipped file containing the following items, and submit it to Blackboard.

1. A report that includes (1) the (printed) full names of the project members, and the statement: **We have neither given nor received unauthorized assistance on this work**; (2) the name of your virtual machine (VM), and the password for the `instructor` account (details see *Accessing VM* below); and (3) a brief description about how you solved the problems and what you 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. The report can be in txt, doc, or pdf format.
2. The source code including both the kernel-level code you added and the user-level testing program.[1]

[1] In case there are issues accessing your VM, submitting your source code separately would allow the instructor to give partial grade.

Accessing VM. Each team should specify the name of your VM that the instructor can login to check the project results. Please create a new user account called `instructor` in your VM, and place your code in the home directory of the instructor account (i.e., `/home/instructor`). Make sure the `instructor` has the appropriate access permission to your code. In your project report, include your password for the `instructor` account.

Tasks

Task 0: Building the Linux kernel (20 points)

Step 1: Getting the Linux kernel code

Before you download and compile the Linux kernel source, make sure that you have development tools installed on your system.

In CentOS, install these software using `yum`:[2]

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

In Ubuntu, install these software using `apt`:

```
# apt-get install -y gcc libncurses5-dev make wget
```

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
```

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
```

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, type this in `LINUX_SOURCE`:

```
$ make localmodconfig
```

Step 3: Compile the kernel

In `LINUX_SOURCE`, compile to create a compressed kernel image:

```
$ make -j8
```

Replace `8` in `make -j8` with the number of cores you have for the VM (typically a number between 4-8. if you don't know, this wouldn't affect you much, just an optimization option).

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
```

[2]The # prompt indicates that the command requires root privilege. $ indicates normal user privilege.

Install the kernel:

```
# export KERNELVERSION=$(make kernelversion)
# cp arch/x86/boot/bzImage /boot/vmlinuz-$KERNELVERSION
# cp System.map /boot/System.map-$KERNELVERSION
# new-kernel-pkg --mkinitrd --install $KERNELVERSION
```

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

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

The kernel image and other related files have been installed into the `/boot` directory.

Step 5: Reboot your VM

Reboot to the new kernel:

```
# reboot
```

After boot, check if you have the new kernel:

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

Task 1: Add a new system call to the Linux kernel (30 points)

In this task, we will 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 called `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 (be careful about the quotation marks – type the `printk` function by hand rather than copy&paste to avoid encoding issues):

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

```
#include <linux/kernel.h>
#include <linux/sched.h>
asmlinkage int sys_helloworld (void)
{
 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/
```

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`. If 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);
```

Repeat step 3 and 4 in task 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 <linux/unistd.h>
#include <sys/syscall.h>
#include <sys/types.h>
#include <stdio.h>
#define __NR_helloworld 337
int main(int argc, char *argv[]) {
 syscall(__NR_helloworld);
 return 0;
}
```

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`.

Task 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`

Task 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 (in Task 2) 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 would be to 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.