CS 3013 Operating Systems WPI, A Term 2016
Craig E. Wills Pre-Project 2 (5 pts)
Assigned: Friday, September 2, 2016 Due: Friday, September 9, 2016

Building a Module and Recompiling Linux

In this pre-project, you will build the Linux environment that you will use in the remainder of the project. In doing so, you will learn how to create, build, install, and remove a Loadable Kernel Module (LKM). You will also add a few system calls to the Linux kernel source code and recompile it. You will then reboot into this new kernel and test out the system call with a user application. This project specification provides significant guidance, making the project a straightforward exercise (and thus worth relatively few points), but it is vital that you complete this preparation to succeed in other projects using it.

This is suggested to be a **team project with two students**, although individual projects are also allowed.

Please note: **the kernel recompilation process can take hours to complete**; please plan accordingly. You may want to do something else while that's churning away. Fortunately, this should be the only time a full recompile is required in this course.

"Hello World" Loadable Kernel Module

For this project, we use Ubuntu 14.04.1 (Trusty Tahr), which is an Ubuntu long-term support (LTS) release. Please use the i386 (x86) version of the OS, even if you have a 64-bit machine, so that you can follow along with these instructions. If you use the x64 version against our advice, you will be expected to know what components need to be changed without staff support. If you are on campus, you may find that `http://cerebro.cs.wpi.edu/ubuntu-14.04.1-desktop-i386.iso` (a file containing what's needed to boot a machine) is a quick mirror that does not count against your CCC bandwidth quota.

VMPlayer Player is available on the Zoo Lab machines in Fuller Labs for creation of a virtual machine. Alternately *VirtualBox* can be installed on your own computer (free versions for Windows, Linux, Macintosh) for creating a virtual machine. We recommend allocating at least 20GB of disk space to the guest VM since a reasonable amount of disk may be used during the kernel compilation process. Once you have installed Ubuntu, you'll want to use the "terminal" application, which is a command shell.

Version note: This assignment is prepared to be used with the Ubuntu 14.04.1 LTS version. As part of installation, you may asked if you want to install a newer version of Ubuntu (16.06.1). It is suggested that you do NOT upgrade as the instructions may not match the new kernel version. If you do upgrade to a newer version then you may need to adjust paths, file names and commands to reflect your system's version numbers.

To prepare for the build, let's make sure we have the latest version of everything for the 16.04.1 version.

Note: The "sudo" command allows a user to execute a command as another user. In the absence of a specific user, it executes as "root". The following commands (as well as others in this handout) need to be issued with sudo command.

- `apt-get update`
- `apt-get upgrade`
- `apt-get dist-upgrade`

If your updates caused a newer version of the kernel to be downloaded and installed, **you should reboot** (`sudo reboot`). If you do not, you may link against an older version of the kernel and build a module that may not work once you reboot.

Next, let's get everything we will need to build a kernel module. Ubuntu actually helps us out with the module-assistant package. We also need to make sure we get the current linux-headers using a cute shell include:

```
apt-get install linux-source module-assistant build-essential linux-headers-$(uname -r)
```

Now, let's start with an example (based on a tutorial by Mark Loiseau):

```
// We need to define __KERNEL__ and MODULE to be in Kernel space
// If they are defined, undefined them and define them again:
#undef __KERNEL__
#undef MODULE

#define __KERNEL__
#define MODULE

// We need to include module.h to get module functions
// while kernel.h gives us the KERN_INFO variable for
// writing to the syslog. Finally, init.h gives us the macros
// for module initialization and destruction (__init and __exit)

#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>

static int __init hello_init(void) {
    // Note the use of ''printk'' rather than ''printf''
    // This is important for kernel-space code
    printk(KERN_INFO "Hello World!\n");

    // We now must return. A value of 0 means that
    // the module loaded successfully. A non-zero
    // value indicates the module did not load.
    return 0;
}

static void __exit hello_cleanup(void) {
    // We are not going to do much here other than
    // print a syslog message.
    printk(KERN_INFO "Cleaning up module.\n");
}

// We have to indicate what functions will be run upon
// module initialization and removal.
module_init(hello_init);
module_exit(hello_cleanup);
```

Take the above code and save it to the file `hello.c` (also available on course Web page). Now, we need to create a `Makefile` (again available on the course Web page) for this code. Remember: Makefiles are sensitive to whitespace. Make sure your indentations use the tab character and not spaces. Finally, if copying from the PDF, make sure dash and equal signs are simple dashes in the file and not formatted special characters. As a convenience these two files are separately available on the course Web site.

```
obj-m := hello.o
KDIR := /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)

all:
	$(MAKE) -C $(KDIR) M=$(PWD) modules

clean:
	$(MAKE) -C $(KDIR) M=$(PWD) clean
```

Once your `Makefile` is ready, run `make`. Your output should look something like the following:

```
drwho@dalek:~/lkm_project> make
make -C /lib/modules/3.13.0.62-generic-pae/build M=/home/drwho/lkm_project modules
make[1]: Entering directory '/usr/src/linux-headers-3.13.0.62-generic-pae'
  CC [M]  /home/drwho/lkm_project/hello.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/drwho/lkm_project/hello.mod.o
  LD [M]  /home/drwho/lkm_project/hello.ko
make[1]: Leaving directory '/usr/src/linux-headers-3.13.0.62-generic-pae'
```

If your output looks similar, your `make` likely completed successfully.

You are now ready to insert the module using `insmod` (short for "insert module"). Naturally, loading kernel modules requires root privileges:

```
drwho@dalek: /lkm_project> sudo insmod hello.ko
```

You should not receive any output (if you receive "-1 Invalid module format," check to make sure your kernel and source code versions are the same). However, take a look at the system log to see how it went:

```
drwho@dalek:~/lkm_project> tail -n 1 /var/log/syslog
Aug  8 15:09:47 dalek kernel: [ 2689.324547] Hello World!
```

The syslog line shows us that the module loaded correctly. If the line is something else, another process could have written to the syslog. You might grep for "Hello World". If it does not appear, your module did not load correctly.

Once you have the module loading correctly, you can remove it again using `sudo rmmod hello.ko` (short for "remove module"):

```
drwho@dalek:~/lkm_project> sudo rmmod hello.ko
drwho@dalek:~/lkm_project> tail -n 1 /var/log/syslog
Aug  8 15:10:04 dalek kernel: [ 2706.044371] Cleaning up module.
```

Again, the syslog output shows us that the removal was successful.

LKMs and Adding a New System Call

While LKMs can add kernel functionality, they interestingly cannot add a new system call, which is the primary mechanism for communicating between user and kernel space. This limitation results from the static definition of system calls using a static array and a fixed system call count definition. Some people claim this is an inherent security defense. These people are clearly misguided: an LKM runs in Kernel space and operates with unchecked power. If you are worried about the security of an LKM, you would be advised to not include it in your kernel.

LKMs actually have the ability to overwrite an existing system call, which is called "interception." The LKM can replace the old system call functionality, augment it, or simply eliminate it. This can certainly be useful, but it can also break user applications when the system call does not behave as expected.

Rather than take an existing system call and intercept it, possibly causing widespread chaos, we will instead create three new system calls by recompiling the Linux kernel. These system calls will initially operate as stubs. We will then alter their functionality by using interception in LKMs. This will allow us to rapidly test and prototype kernel level code, without continually recompiling the kernel and rebooting. It also means that if we do something horribly wrong, we can simply reboot the system and the errant module will not be loaded.

But, before we do that, we must add the system calls to the kernel, recompile, and boot into the modified kernel. Joy.

Recompiling the Kernel in Ubuntu

We begin with some preliminaries:

```
sudo apt-get install kernel-package libncurses5-dev fakeroot wget bzip2
```

Earlier, we installed linux-source, which provides the `/usr/src/linux-source-3.13.0` directory. Upon entering that directory, run `sudo tar -jxvf linux-source-3.13.0.tar.bz2`, which will take a little bit and extract all the source you need. For convenience, we'll create a symbolic link to this extracted source:

```
sudo ln -s /usr/src/linux-source-3.13.0/linux-source-3.13.0 /usr/src/linux
```

Note: The "-s" option of the ln command creates a symbolic link, which is an "alias" or "shortcut" for the filename.

We now enter `/usr/src/linux` and run `sudo cp /boot/config-$(uname -r) ./.config`. This command copies the configuration of the currently running kernel to the .config file in the current directory. This will make it easier for us to create our own kernel variant with the same functionality of the current kernel.

Editing the Kernel Source Code

We are now going to edit the source code for the kernel. Most people make a copy of the source before making edits, but we're going to make the edits directly since we're only making minor changes on a VM. Naturally, you will want to make backups of any files you will alter before you edit them. If you do not, it may be exceedingly difficult for you to undo any errors.

To backup our files, we are going to touch, we'll do the following as root:

- `cp /usr/src/linux/kernel/sys.c /usr/src/linux/kernel/sys.c_backup`
- `cp /usr/src/linux/arch/x86/include/asm/unistd_32.h /usr/src/linux/arch/x86/include/asm/unistd_32.h_backup`
- `cp /usr/src/linux/arch/x86/kernel/syscall_table_32.S /usr/src/linux/arch/x86/kernel/syscall_table_32.S_backup`

We will now edit each of these files in turn to add three system calls. Please follow these instructions EXACTLY and recognize that line order matters in several places.

Edit `/usr/src/linux/kernel/sys.c` to add the following lines to the end of the file:

```
SYSCALL_DEFINE0(cs3013_syscall1) {
        printk(KERN_EMERG "'Hello, world!' -- syscall1\n");
        return 0;
}
SYSCALL_DEFINE0(cs3013_syscall2) {
        printk(KERN_EMERG "'Hello, world!' -- syscall2\n");
        return 0;
}
SYSCALL_DEFINE0(cs3013_syscall3) {
        printk(KERN_EMERG "'Hello, world!' -- syscall3\n");
        return 0;
}
```

Be sure to include a newline ('\n') character after the last '}' character to avoid compilation errors. Also, note that there is no comma between `KERN_EMERG` and `"'Hello` in this code.

We now edit `/usr/src/linux/arch/x86/syscalls/syscall_32.tbl` to define each of these system calls. Around line 364, you will see the end of a list of defined system calls. We now add our three, incrementing the count for each (again, use tabs as delimiters):

```
355     i386      cs3013_syscall1      sys_cs3013_syscall1
356     i386      cs3013_syscall2      sys_cs3013_syscall2
357     i386      cs3013_syscall3      sys_cs3013_syscall3
```

Finally, we edit `/usr/src/linux/include/linux/syscalls.h`. Go to the end of the file. The last line is `#endif`. **Before** that line, add the following three lines:

```
asmlinkage long sys_cs3013_syscall1(void);
asmlinkage long sys_cs3013_syscall2(void);
asmlinkage long sys_cs3013_syscall3(void);
```

Building the Kernel

We now have to set those configuration settings by running `sudo make menuconfig` (or for those who like GUIs, `sudo make xconfig`) in `/usr/src/linux`. You will be greeted with an ncurses menu (or GUI window) for selecting options to configure the kernel. We could make a bunch of hard decisions ourselves. But, instead, we'll use the .config file we just saved by going down to the "Load an Alternative Configuration File" option and selecting it. It will automatically fill in the value with ".config", which is what we want. Select "ok."

We now have the option to leave our mark in the kernel name. This will help us identify our kernel in the boot loader, amongst other things. Go under "General setup" and select the "Local version - append to kernel release" option and put in your WPI username into the box. Be sure NOT to use any characters other than letters or numbers since they may cause extremely confusing compilation errors. If you have non-alphanumeric characters in your username, simply omit them. When done, choose "ok" and then "exit" to the main menu. Once there, choose "exit" again and say "yes" to save the configuration.

Now, compile the kernel. Be sure you are in `/usr/src/linux` and then run:

```
sudo make-kpkg clean
export CONCURRENCY_LEVEL=2
sudo fakeroot make-kpkg --initrd kernel_image kernel_headers
```

The third command, above, is going to take a while. Depending on your CPU, this step could take several hours. One instructor's compilation took 2 hours and 10 seconds on an old MacBook Pro using the Parallels VMM with the guest VM having 1 CPU and 1GB memory. His host OS was still usable, but one of his cores was pegged. If you have a multi-core system and have assigned multiple cores to your guest VM, you can increase the CONCURRENCY_LEVEL variable above to increase your compilation speed (quite dramatically, in some cases). Using a 16 core server with 64GB RAM, the instructor provided the guest VM with 10 cores and 8GB of RAM and set CONCURRENCY_LEVEL=11. The compilation took 40 minutes, and 15 seconds. Another compilation on a Windows desktop machine with 1GB of memory dedicated to the virtual machine took approximately 1.5 hours to complete. Yet another on a similar machine took close to 3 hours.

Now, cd into `/usr/src/linux-source-3.13.0`. Perform a listing to determine your .deb file names (these files are software packages used in Linux distributions based on Debian). Then, as root, use `dpkg` to install the image .deb and then the headers .deb file. You should find (and use) similar names as:

```
sudo dpkg -i linux-image-3.3.11-ckt24cew_3.13.11-ckt24cew-10.00.Custom_i386.deb
sudo dpkg -i linux-headers-3.3.11-ckt24cew_3.13.11-ckt24cew-10.00.Custom_i386.deb
```

These packages will update your grub2 boot loader files, amongst other things. You will be rebooting the system shortly and may want to update a grub2 setting to make it easier to boot your modified kernel. To do so, edit `/etc/default/grub` to comment out the line `GRUB_HIDDEN_TIMEOUT=0` by prepending it with a # character. Then, do `sudo update-grub` to update the boot loader. Alternatively, when booting your kernel, you can get the Grub2 boot menu by holding down the right shift key (though the timing is hard to master). Once you've made your preparations, or practiced your typing reflexes, reboot the VM by running `sudo reboot` or `sudo shutdown -r now`.

Once you get to the Grub2 screen, select "Advanced Ubuntu Options" and then select your version of the kernel (the one with your username in it).

Once your boot process has completed, you can test to see whether the added systems call work by using the following code (available on the course Web site as `proj2test.c`):

```
#include <sys/syscall.h>
#include <stdio.h>

// These values MUST match the syscall_32.tbl modifications:
#define __NR_cs3013_syscall1 355
#define __NR_cs3013_syscall2 356
#define __NR_cs3013_syscall3 357

long testCall1 ( void) {
        return (long) syscall(__NR_cs3013_syscall1);
}
long testCall2 ( void) {
        return (long) syscall(__NR_cs3013_syscall2);
}
long testCall3 ( void) {
        return (long) syscall(__NR_cs3013_syscall3);
}
int main () {
        printf("The return values of the system calls are:\n");
        printf("\tcs3013_syscall1: %ld\n", testCall1());
        printf("\tcs3013_syscall2: %ld\n", testCall2());
        printf("\tcs3013_syscall3: %ld\n", testCall3());
        return 0;
}
```

Once you compile and run the code, you should receive return values of 0. However, you should also see greetings from each of the system calls in `/var/log/syslog`. If you see your hello world messages, you know that you have properly created your system calls and invoked them. We'll then be able to modify these system calls in future projects.

Assignment Submission and Deliverables

Be sure to put all team members at the top of each of the files that you submit. All team members should be familiar with building and booting a kernel. When submitting your project, please include the following:

- A text file containing the output of the shell command `uname -a` on your modified kernel.
- The contents of your modified `sys.c`, `syscall_table_32.S`, and `unistd_32.h` files.
- The contents of your `testcalls.c` file used to evaluate the program.
- The contents of your `/var/log/syslog` file.

Please upload your .zip archive via Web turnin with the project name of *preproj2*.

References

- Mark Loiseau, "'Hello World' Loadable Kernel Module," `http://blog.markloiseau.com/2012/04/hello-world-loadable-kernel-module-tutorial/`, April 2012 *(currently offline)*
- Bryan Henderson, "Linux Loadable Kernel Module HOWTO," `http://www.tldp.org/HOWTO/html_single/Module-HOWTO/`, September 2006.
- nixCraft, "HowTo: Ubuntu Linux Install Kernel Source Code And Headers," `http://www.howtoforge.com/kernel_compilation_ubuntu`, July 2009.