CS-3013, Operating Systems
C-Term 2018

Project 0 (20 points)
Assigned: Friday, January 12, 2018
Due: Thursday, January 18, 2018

Project 0: Building a Module and Recompiling Linux[©]

Hugh C. Lauer, Teaching Professor
Robert J. Walls, Assistant Professor
Worcester Polytechnic Institute

In this project, you will build the Linux environment that you will use in a subsequent course 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 future projects.

This is an *individual project*. Although you may help each other, each student should eventually carry out all the steps himself or herself to build and install a Linux kernel.

Notes: It is essential that you start this project as soon as possible and not wait till near the due date.

Enough has changed about building Linux kernels in Ubuntu that there may be lingering surprises that could consume a lot of time.

Normally, the Linux kernel takes one or more hours to build from source files. The configuration that has been prepared for this assignment take substantially less time to build, but it results in a simpler system with fewer conveniences that we have become accustomed to.

On the other hand, the Linux source files still do take a long time to download from **git**. You should carry out this part of the project on campus or at a location where you have adequate (and affordable) download bandwidth.

Getting Ready

By now, you should be accustomed to using a *virtual machine* for course-work in Computer Science courses at WPI. Fetch and download a virtual machine and instructions from:–

http://cs.wpi.edu/~cs3013/c18/Resources/CS-3013_Virtual_Machine.ova
http://cs.wpi.edu/~cs3013/c18/Resources/SettingUpYourVirtualMachine.docx, pdf

This file is a 2.6 gigabyte file. If you live off campus, you should download it on a public computer *on campus* to a USB flash drive and then carry it to your own computer off campus.

[©] Copyright 2018, Hugh C. Lauer. All rights reserved. Permission is given for use in courses at Worcester Polytechnic Institute, Worcester, Massachusetts.

Once downloaded, set up this virtual machine on your own Windows, Macintosh, or Linux laptop or desktop computer as described in the document using the free *Virtual Box* application.

If you already have a Linux virtual machine — for example, from a course in a previous term — please set that one aside and use this one.

Also, as described in the document, adjust the number of processors to at least half of the number of processors on your host machine, and adjust the amount of virtual RAM to at least half the amount of physical RAM on your host machine. Finally, re-install the Virtual Box Guest additions.

Downloading and Building the Linux Kernel

In previous terms, WPI students were able to download the Linux kernel on Ubuntu virtual machines using the **apt-get** utility and then build the kernel using a simple set of **make** commands. Since that time, the Ubuntu configuration and tools for compiling and building the kernel have changed, and the previous CS-3013 instructions do not seem to work anymore. Therefore, here is the outline of this project:–

- Download the Linux kernel sources using **git**
- Configure, compile, and test the kernel
- Create and install a new Loadable Kernel Module (LKM)
- Test the new LKM
- Submit the results to *InstructAssist*

Downloading

Open a command shell by typing CTRL-ALT-T. In the command shell, execute the following commands to install useful stuff (ignore the comments after the '**//**'):–

```
sudo apt update                 // we already updated ubuntu, but just make sure
sudo apt install build-essential            //building tools for Linux
sudo apt install libncurses5-dev gcc make git exuberant-ctags bc libssl-dev libelf-dev
```

For the main part of the build, work in a directory with ordinary user permissions. Execute

```
git clone git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-stable.git
```

This fetches the sources for the Linux kernel from the **kernel.org**. It creates the directory **linux-stable** and fetches about 6.4 million objects; this takes about six minutes on campus.

Configuring and compiling

Change to the directory **linux-stable**. Execute the following commands with ordinary user privileges.

```
make defconfig              // Creates the default configuration for this particular version
                            // of Ubuntu Linux. Result is stored in the file .config

make localmodconfig         // Modifies .config to include only the kernel modules actually
                            // loaded into your running kernel. I.e., strips out a lot of
                            // extraneous stuff

make menuconfig
```

The **make menuconfig** command is a general purpose configuration tool for controlling what should be included in the kernel. It brings up a pseudo GUI in which you need to make a single change. *Using the arrow keys*, scroll down to the second line, labeled **General setup**. Type the **Enter** key to select this. In the next window, scroll down to and select the third line labeled **Local version – append to kernel release**. Type **Enter** to select it; a "window" called **Local version – append to kernel** will pop up. Enter your real name or your user name and type **Enter** to accept it.[1]

This returns you to the **General Setup** "window." Using the left and right arrow keys, select the **<Save>** command at the bottom of the window and type **Enter**. Finally, confirm your selection and then scroll to the **<Exit>** command and type **Enter** to finish the configuration.

You are now ready to compile. Type the following (with ordinary user privileges): –

```
make –j4          // replace '4' with number of processors in your virtual machine
```

This compiles the kernel. In the "old days," this would take an hour or more. Using **make localmodconfig** reduces the number of modules to compile, so this completes in about 5-7 minutes on a medium speed Core i7. The build finishes with the message "**Kernel: arch/x86/boot/bzImage is ready**."[2]

Finally, to install the new kernel, execute

```
sudo make modules_install install
```
[3]

To actually run the new kernel, type the command

```
sudo reboot
```

As soon as you seen the Oracle Virtual Box splash screen, immediately press and hold down the left-shift key to cause the "GRUB" boot window to appear. Use the arrow keys to scroll up or down to **Advanced options for Ubuntu**. Select this with the Enter key, and then scroll to the kernel with your name on it.[4]

Open a command prompt and type the command

```
uname –r
```

This should display a kernel identification string with the name you created in **make menuconfig** above.

Take a few minutes to discover what you can do with this kernel.

Create a Loadable Kernel Module (LKM)

A *module* in Linux is a compiled program that can be configured into or left out of the Linux kernel, thereby including or excluding a particular function or feature. Indeed, the configuration steps in the kernel build process amount to a selection of which modules to include and which to leave out. A *Loadable Kernel Module* is simply a Linux module that can be compiled separately and loaded at run-time. It has all of the capabilities and privileges of modules included at kernel build time.

1 This name may be your real name or your WPI user ID. We need to see that it is actually your kernel.
2 Even though the directory is named "x86," the configuration compile for the x86_64 architecture.
3 Note that the Virtual Box "Guest Additions" produce an error message and are not loaded into the new kernel.
4 This kernel will likely have a smaller screen and possibly some other limitations.

We will start with a simple example based on a tutorial by Mark Loiseau,[5] consisting of the following code. Copy or download this into a suitable working directory name it **hello.c**. This program will be your kernel module; the only thing it does is to add an entry to the system log when it is loaded and another one when it is unloaded.

```
// 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);
```

A copy of this can be found at

http://cs.wpi.edu/~cs3013/c18/Resources/cs3013_project0.c

Save a copy of this code as the file **hello.c**.

[5] The original reference to the tutorial containing this module is http://blog.markloiseau.com/2012/04/hello-world-loadable-kernel-module-tutorial/, dated April 2012. Unfortunately, this URL appears to be no longer valid.

In addition, you need a **`Makefile`** to compile this code. This can be found at

http://cs.wpi.edu/~cs3013/c18/Resources/cs3013_project0_Makefile

Download this to the file **`Makefile`** (with a capital 'M'). If you examine it, you will find that it contains the following

```
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
```

Remember that **`makefiles`** are sensitive to whitespace. Your indentations must be tab characters, not spaces; everything that looks like a dash must be a simple hyphen character.

Once your **`Makefile`** is read, run **`make`**. You can see from the output that this creates the file **`hello.o,`** and from that, it creates the kernel module **`hello.ko`**.

You are now ready to insert the module using the command **`insmod`** (short for "insert module"). Naturally, this requires root privileges,

```
sudo insmod hello.ko
```

Likewise, you can remove a module with the **`rmmod`** command. Although neither command produces output on **`stdout`**, both leave trails in the system log, as shown in the following example:–

```
student@CS-3013-VM:~/LKM$ sudo insmod hello.ko
student@CS-3013-VM:~/LKM$ tail -n 1 /var/log/syslog
Jan  9 20:42:37 CS-3013-VM kernel: [19352.055376] Hello World!

student@CS-3013-VM:~/LKM$ sudo rmmod hello.ko
student@CS-3013-VM:~/LKM$ tail -n 1 /var/log/syslog
Jan  9 20:42:52 CS-3013-VM kernel: [19366.782210] Cleaning up module.
```

Adding System Calls to the Kernel

To add a system call to the kernel, we need to edit the source code of the kernel we built earlier in this assignment under the section entitle "Configuring and compiling" above. It is useful to backup each of the four files to be edited before actually editing them.

Set your working directory to **linux-stable** — i.e., the Linux source tree from above. For each of the following edits, it is recommended that you back up the file first.[6]

First edit the file **kernel/sys.c** to add the following lines to the end of the file:–

```
SYSCALL_DEFINE0(cs3013_syscall1) {
	printk(KERN_EMERG "'Hello, world #1!' -- syscall1\n");
	return 0;
}
```

[6] If you were doing this to a live Linux installation, backing up would be a *mandatory* safety step.

```
SYSCALL_DEFINE0(cs3013_syscall2) {
     printk(KERN_EMERG "'Hello, world #2!' - syscall2\n");
     return 0;
}
SYSCALL_DEFINE0(cs3013_syscall3) {
     printk(KERN_EMERG "'Hello, world #3!' - syscall3\n");
     return 0;
}
```

Be sure to include a newline character ('\n') after the last '}' character to avoid compilations errors. Also note that there is NO comma between KERN_EMERG and the "Hello World" string.

Next, edit the file **arch/x86/entry/syscalls/syscall_32.tbl** to define each of these system calls. Find the last system call in this table (approximately #384) and add three lines for three new system calls using the next numbers in sequence:–

```
385[7] i386 cs3013_syscall1 sys_cs3013_syscall1
386    i386 cs3013_syscall2 sys_cs3013_syscall2
387    i386 cs3013_syscall3 sys_cs3013_syscall3
```

Be sure to use tab characters for delimiters between the fields of each line.

Also edit the file **arch/x86/entry/syscalls/syscall_64.tbl** to define the same system calls in the 64-bit kernel. You will see that the numbers are different. Scroll down to the end of the list of "common" system calls — i.e., approximately after #332. Add the following lines, with the numbers adjusted for your sequence and using tab characters to delimit the fields:–

```
333  common    cs3013_syscall1 sys_cs3013_syscall1
334  common    cs3013_syscall2 sys_cs3013_syscall2
335  common    cs3013_syscall3 sys_cs3013_syscall3
```

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

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

Next, use make menuconfig to put a different label on this kernel, and then rebuild it. This is likely to take as long as the first time that you built it, because the most of the kernel files are dependent upon one or more of the files that you just edited.

Install your new kernel using **make modules_install install**, reboot, and hold down the left shift key when the Oracle Virtual Box splash screen appears. Select the new kernel to test.

Testing your system calls

To test your system calls, reboot your virtual machine, hold down the left-shift key when the Oracle Virtual Box splash screen appears, and then select the modified kernel to boot. Write and compile a user-space program along the following lines:–

[7] These numbers are liable to change with new updates to the Linux kernel.

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

#define __NR_cs3013_syscall1 333
#define __NR_cs3013_syscall2 334
#define __NR_cs3013_syscall3 335

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());
}
```

Note that the system call numbers are those you defined in **`syscall_64.tbl`**, because we are working with 64-bit Linux kernels.

Compile and run this program to actually test your system calls. The output should be something along the following lines:–

```
Jan 10 15:13:38 CS-3013-VM kernel: [   41.240878] 'Hello, World!'
-- syscall1
Jan 10 15:13:38 CS-3013-VM kernel: [   41.240929] 'Hello, World!'
-- syscall2
Jan 10 15:13:38 CS-3013-VM kernel: [   41.240940] 'Hello, World!'
-- syscall3
```

Checkpoint Contributions

Project 0 (this project) does not have a checkpoint deliverable.

Assignment Submission and Deliverables

Be sure to put your name at the top of each of the files that you create or edit. When submitting your project, please create a zip file that includes the following files: –

- Your **`hello.c`** and your **`Makefile`** programs for the LKM portion of the project.
- A text file containing the output of the shell command **`uname -a`** on your modified kernel.

- The contents of your modified **`sys.c`**, **`syscall_32.tbl`**, **`syscall_64.tbl`**, and **`syscalls.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 InstructAssist (https://ia.wpi.edu/cs3013/files.php) in the "Project 0" folder.

References

- Mark Loiseau, "'Hello World' Loadable Kernel Module," http://blog.markloiseau.com/2012/04/hello-world-loadable-kernel-module-tutorial/, April 2012.
- 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.