Lab 3: Loadable Kernel Modules

This lab assignment consists of two parts.
Part 1: building the Linux kernel, you will:

1. Download a custom version of the Linux source code for the Raspberry Pi 3
2. Configure options to further customize a Linux kernel
3. Compile the kernel source using those options to produce files for installation
4. Install, and boot up your new custom kernel

Part 2: writing simple kernel modules, you will:

5. Build and install kernel modules
6. Write a kernel module and use it to observe changes in a kernel variable

---

You will need Raspberry Pi (or a Linux machine) to complete the lab. The instructions below are for using Raspberry Pi. If you use other Linux machine or virtual machine to complete this lab, you will need to find the proper commands (some may be different from the instructions) to complete Part 1. No matter what machine you use, your goal is to complete the required exercises highlighted in red and record your answers to the exercises in a report.

Make sure that the name of each person who worked on these exercises is listed in the first answer, and make sure you number each of your responses so it is easy to match your responses with each exercise.

This lab has 4 points in total.

Submission:

When finished, ***submit your report in the PDF form to Moodle***. In addition to the report, please also submit the jiffies_module.c file for the new kernel module you created.

Required Exercises:

0. As the answer to the first exercise, list the names of the team members working on this lab.

1. Boot up and log into your Raspberry Pi, open up a terminal window, and use the uname command to get a variety of information about the currently running system, such as the current kernel version or the date on which the currently running kernel was compiled. As the answer to this exercise, give the output from running the command uname -a.

2. Now it's time to download the Linux kernel source code. For a general Linux machine, you would go to kernel.org and download the Linux source from there. However, since we're using the Raspberry Pi we'll be starting with a version that is designed for our platform. The Raspberry Pi project maintains it's own distribution on GitHub, at https://github.com/raspberrypi.

There are two main methods for building the kernel. You can build locally on a Raspberry Pi, which will take a long time; or you can cross-compile, which is much quicker, but requires more setup and can be difficult to beginner. We will follow Raspberry Pi project's official guide at https://www.raspberrypi.org/documentation/linux/kernel/building.md to build the kernel.

Please refer to the official guide for detailed instructions on kernel building on Raspberry Pi. Below summarize the important steps and commands for local compiling on Raspberry Pi:

- Prepare and down load the proper version of Linux source:
  - `sudo apt-get install git bc`
  - `git clone --depth=1 https://github.com/raspberrypi/linux`
- Configure the kernel (IMPORTANT: the commands below differ on different machine, see the official guide for details):
  - `cd linux`
  - `KERNEL=kernel7`
  - `make bcm2709_defconfig`
- Set a custom configuration option:
  - `make menuconfig`
  - Hint: if the above command returns any error message, then install the missing packages based on the message
  - After a moment, you'll get a kernel configuration menu. You are asked to add your own unique identifier to the kernel you build. Navigate to "Local version" under "General setup". The local version string you specify here will be appended to the output of the uname command. If you applied the default Raspberry Pi configuration correctly, this should be set to "-v7". Go ahead and append your own unique identifier (e.g. "CS630") after the current string.
  - **Warning: do not include a space in the local version string- this will break the build script when you run sudo make modules_install**.
- Build and install the kernel:
  - `make -j4 zImage modules dtbs`
  - `KERNEL=kernel7` (if you've logged out since issuing this command before)
  - `sudo make modules_install`
  - `sudo cp arch/arm/boot/dts/*.dtb /boot/`
  - `sudo cp arch/arm/boot/dts/overlays/*.dtb* /boot/overlays/`
  - `sudo cp arch/arm/boot/dts/overlays/README /boot/overlays/`
  - `sudo cp arch/arm/boot/zImage /boot/$KERNEL.img`
  - Hint: this step takes a long long time, so make sure your Pi has a good and stable power supply. If your Pi crashes during the `make` step, then you may want to try reducing the number of cores used by changing "`-j4`" to "`-j3`" or even fewer.
- At this point, your new kernel is installed. When you reboot, you'll be running your very own, custom kernel. Go ahead and reboot now. If everything went OK, the new system should look and feel the same as before.

- You can verify that your new kernel is running with the command uname -a. Congratulations! You've just compiled an operating system from source!

As the answer to this exercise, give the output from running the command uname -a.

Note: from now on (i.e., in Part 2), the instructions are the same no matter which Linux machine or Raspberry Pi you are using. So the remaining exercises only highlight the requirement for answers in the report or code submissions.

3. In the linux_source directory (which contains the linux directory in which you've built the kernel), create a new directory to hold your kernel modules, and cd into it. Save a copy of the provided `simple_module.c` (a simple template for writing kernel modules in this course, based on the "Hello, World!" module shown on pages 338 and 339 of Robert Love's Linux Kernel Development, Third Edition) and the provided `Makefile` into that directory.

Now you can compile the module by issuing the command:

```
make
```

Before loading the module, clear out the contents of the system log using the command:

```
sudo dmesg --clear
```

Then use the `insmod` utility to load your kernel module into the kernel, as in:

```
sudo insmod simple_module.ko
```

If you received no error messages, then your module has been successfully loaded.
To confirm your module was loaded, you can also issue the command

```
lsmod
```

To confirm it for the specific module, you can also check the system log by issuing the command

```
dmesg
```

which prints out the system log, which now should show the message that was printed when the module loaded.
The basic utility for removing modules from the kernel is called `rmmod`. When using this tool you can either specify the module name (as shown in lsmod) or you can specify a .ko file, as in:

```
sudo rmmod simple_module.ko
```

Remove your module now, and verify its removal as before. As the answer to this exercise, copy the output of `dmesg` after removing the module.

4. One major reason for using kernel modules (as opposed to running a userspace program with root permissions) is that module code has direct access to all of the kernel's resources. Userspace programs, in contrast, must use system calls to access such resources, and even then, most of the kernel remains opaque to user processes.

One kernel variable we are going to use here is the `jiffies` counter. This variable keeps track of how many timer interrupts (also called *ticks*) have occured since system boot. See this webpage for more explanations: https://cyberglory.wordpress.com/2011/08/21/jiffies-in-linux-kernel/

Copy `simple_module.c` to a new file called `jiffies_module.c` and modify that new file, so that the system log messages that are generated when the module is loaded and unloaded also give the value of the `jiffies` variable (hint: it's an unsigned long) to the system log. Note that although this `jiffies` variable is not readily available to userspace programs, it is available directly when in kernelspace.

Update your `Makefile`, build your new kernel module. Load and unload your new kernel module. As the answer to this exercise please copy and paste the system log message that shows the values of the jiffies variable when your module was loaded and when it was unloaded, and say how many ticks occurred between those messages.