Laboratory Assignment #1

Warm Up: Compiling a Kernel and a Kernel Module

Value: *(See the **Grading** section of the Syllabus.)*

Due Date and Time: *(See the **Course Calendar**.)*

Summary:
This is not exactly a programming exercise, since you will not be writing any new code. You will compile and load a kernel starting from the existing Linux "vanilla" sources. The basic steps are: configure, compile, install, and test. You will then do the same for a trivial kernel module, given in this text. This will make sure you can handle the mechanics, let you see first-hand the time scales of some of the development steps you will be repeating frequently throughout the course, and expose you to some of the many possible failure modes of a kernel module.

Objectives:
- Learn a few Linux system administration commands, the bare minimum you need to compile, install, and test a kernel.
- Learn to reduce boot time and compile time by turning off unneeded Linux system services, and by pruning kernel options.
- Experience configuring, compiling, and installing a Linux kernel.
- Experience compiling and installing a set of Linux kernel modules.
- Experience kernel errors.

Tasks:
1) Maintain a journal documenting the steps taken to complete this assignment.

2) Configure and compile a "minimal" but bootable Linux kernel image using the Linux "vanilla" source code with a version >= 3.10. The bootable image must not use any kernel modules and must be less than 5MiB in size. Document the size of the kernel image in your journal.

3) Successfully boot your kernel image without using an initramfs (initrd).

4) Build and install the hello.ko kernel module.

5) Generate at least one kernel error by modifying the hello kernel module source code. Document the error and the resulting kernel output.

6) Demonstrate and answer questions about the above tasks.

7) If you are using a machines in the lab (MCH 202):
    - Find a machine that no one else has yet claimed.
        Take care to check that it has a parallel port, since you will be using the same machine for assignment #4, which requires a parallel port.
    - On a piece of paper or Post-It, write your name and tape it to the monitor.

Delivery Method:

1. In class on the due date, arrange an appointment with the instructor/TA to demonstrate what you have done.
2. At the appointment, come to the lab and demonstrate your work. At that time, turn in on paper your journal/notes of the "bad" modules you tested, including copies of the code, what behavior they caused (*e.g.*, system locked up, kernel panic), and what output (if any) was generated. Be prepared to demonstrate your kernel booting up and your modules loading and executing, explain what steps you took, walk through your configuration option selections, and demonstrate the module(s) that caused crashes. Have the test modules ready to go, in separate source files. *Do not edit code during the demonstration.*

Assessment:

If you do everything that is required and can explain it adequately during the demonstration you will receive a perfect score. These include

1. Kernel image smaller than 5MiB.
2. No modules required to boot the system.
3. No initramfs required to boot the system.
4. New kernel added to grub.
5. “hello” kernel module compiles in new kernel.
6. Misbehaving module created.
7. Assignment journal maintained.

Deductions will be made for:

1. Uncompleted steps.
2. Inability to explain what you have done.
3. A large kernel:
    a. Size of kernel image over 5MiB.
    b. Modules loaded as shown by lsmod and by applying the du command to the subdirectory of /lib/modules/... for your kernel version.
4. Having to edit code, recompile, etc. during demonstration of kernel failure modes.

General Steps and Information:

Overview

The following is a general guide to compile a Linux kernel image from source code, install the compiled kernel, and boot the system. These steps assume a particular version of the Linux kernel source tree (3.10.39) and a particular Linux distribution (Linux Mint 13). If you are using a

different Linux distribution you will need to adapt the instructions, or install a compatible distribution. The instructions in the guide explain one way of doing the job. If you have built Linux kernels before, you are free to compile the kernel in your own way, so long as it works. If you catch anything that looks like a typo or omission in the description, please let me know.

Some recommendations:

- Get started early so there is time to ask the instructor or TA about any questions you may have.
- Start with vanilla kernel version 3.2.58. Once you have a .config that works with 3.2.58, run make oldconfig on the 3.10 sources.
- Be prepared for the possibility that your machine's contents may be corrupted by backing up your work.
- Separate your personal files (in /home) from the OS files (in /), using different disk partitions, so that if you need to reload the OS you will not need to overwrite your home directory.

User Accounts Setup

Log in as the "device" user. The device user is setup to allow sudo, which allows users to run programs with escalated user privileges. You may also want to create a "guest" account for use by other students, later, if you set your machines up in pairs for remote console message logging using the serial ports. Perform the following steps to add a user account to the system:

- sudo useradd -m -G sudo <your choice of userid> will create a new user and setup the associated /home/… directory structure
- Members of the sudo group are already sudoers, so the following is not necessary but is left for your information. sudo visudo can be used to add the user you created above to the sudoers list, effectively, giving the user root access.
- sudo su (or sudo -s) is used to switch to the root user

I recommend that you only do as root those things that require it, to reduce the risk of clobbering your system if you make a mistake. In general, it is wise to log in first as a normal user, and then just "sudo su" to root temporarily when necessary, or to use one of the Linux "virtual consoles" for a separate root login that you use just for the commands that require it. Yet another option is to use sudo to run individual commands as root.

Faster boot times

Disable some of the unneeded services. You will need to exercise a little bit of trial and error, and some web searching on the names of the services, to see which ones you can do without. Test the system with the reduced services, by rebooting, and verifying that the system is working well enough for you to continue with the rest of this exercise. You can use the command "reboot" to reboot the system. **Do not power-cycle your system to reboot, if possible**. It may prematurely age your hardware.

The Linux Kernel

You may need to install additional packages to compile and build a kernel. Assuming you have networking up and running, you can do this using the apt-get utility. To install individual packages, run sudo apt-get install <package name>. For building a Linux kernel you may need the following packages: libncurses5-dev. You can read more about *apt-get* in the man-page. Also, *apt-cache* can be used to search for packages.

As the normal user to whose home directory you copied the kernel source code, extract the kernel source tree from the archive file and put in a location of your choice. For example, if you followed the steps above, you could next do the following:

```
cd /home/user
tar --xz -xpf linux-3.10.39.tar.xz
```

- Once you have the kernel source tree uncompressed and untarred, cd into the source directory and configure the kernel, as follows:

  ```
  cd linux
  make menuconfig
  ```

- The kernel comes in a default configuration, determined by the people who put together the kernel source code distribution. It will include support for nearly everything, since it is intended for general use, and is *huge*. In this form it will take a very long time to compile and a long time to load. For use in this course, you want a different kernel configuration. The "make menuconfig" step allows you to choose that configuration. It will present you with a series of menus, from which you will choose the options you want to include. For most options you have three choices: (blank) leave it out; (M) compile it as a module, which will only be loaded if the feature is needed; (*) compile it into monolithically into the kernel, so it will always be there from the time the kernel first loads.
- The .config for the currently running kernel is in the /boot directory and can be used to get an initial kernel running, but be aware that the default .config enables many unneeded options and therefore building a kernel will take a very long time (> 1 hour).
- There are several things you want to accomplish with your reconfiguration:
  - Reduce the size of the kernel, by leaving out unnecessary components. This is helpful for kernel development. A small kernel will take a lot less time to compile and less time to load. It will also leave more memory for you to use, resulting in less page swapping and faster compilations.
  - Retain the modules necessary to use the hardware installed on your system. To do this without including just about everything conceivable, you need figure out what hardware is installed on your system. You can find out about that in several ways.
- Before you go too far, use the "General Setup" menu and the "Local version" and "Automatically append version info" options to add a suffix to the name of your kernel, so

that you can distinguish it from the "vanilla" one. You may want to vary the local version string, for different configurations that you try, to distinguish them also.

- If you have a running Linux system with a working kernel, there are several places you can look for information about what devices you have, and what drivers are running.
    - Look at the system log file, /var/log/messages or use the command dmesg to see the messages printed out by the device drivers as they came up.
    - Use the command lspci -vv to list out the hardware devices that use the PCI bus.
    - Use the command lsusb -vv to list out the hardware devices that use the USB.
    - Use the command lsmod to see which kernel modules are in use.
    - Look at /proc/modules to see another view of the modules that are in use.
    - Look at /proc/devices to see devices the system has recognized.
    - Look at /proc/cpuinfo to see what kind of CPU you have.
    - Look at /proc/meminfo to see how much memory you have.
    - Check the hardware documentation for your system. If you know the motherboard, you should be able to look up the manual, which will tell you about the on-board devices.

- Using the available information and common sense, select a reasonable set of kernel configuration options. Along the way, read through the on-line help descriptions (for at least all the top-level menu options) so that you become familiar with the range of drivers and software components in the Linux kernel.
- Before exiting the final menu level and saving the configuration, it is a good idea to save it to a named file, using the "Save Configuration to an Alternate File" option. By saving different configurations under different names you can reload a configuration without going through all the menu options again. Alternatively, you can back up the file (which is named .config manually, by making a copy with an appropriate name.
- One way to reduce frustration in the kernel trimming process (which may involve quite a bit of trial and error) is to start with a kernel that works, trim just a little at a time, and test at each stage, saving copies of the .config file along the way so that you can back up when you run into trouble. However, the first few steps of this process will take a long time since you will be compiling a kernel with huge number of modules, nearly all of which you do not need. So, you may be tempted to try eliminating a large number of options from the start. *Note: There is a new command introduced since 2.6.32 that may help (or hurt) this process – read http://www.h-online.com/open/features/Good-and-quick-kernel-configuration-creation-1403046.html to learn about it and the potential help it could provide.*
- Run the following *make* commands:

  make
  make modules_install
  make install

    - The first command will compile the kernel and create a compressed binary image of the kernel. After the first step, the kernel image can be found at

/boot/vmlinuz-[kernel_version]. The second command will install the dynamically loadable kernel modules in a subdirectory of "/lib/modules", named after the kernel version. The resulting modules have the suffix ".ko". For example, if you chose to compile the network device driver for the Realtek 8139 card as a module, there will be a kernel module name 8139too.ko. Note that for this assignment no modules should be used for booting. The third command is OS specific and will copy the new kernel into the directory "/boot".

  - If there are error messages from any of the *make* stages, you may be able to solve them by going back and playing with the configuration options. Some options require other options or cannot be used in conjunction with some other options. These dependencies and conflicts may not all be accounted-for in the Linux configuration program. If you run into this sort of problem, use the error message to find the missing dependency. For example, if the linker complains about a missing definition of some symbol in some module, you might either turn on an option that seems likely to provide a definition for the missing symbol, or turn off the option that made reference to the symbol.

- An initramfs may be helpful later in the course and so is discussed here so you can become familiar with it. However, for demonstration of this assignment an initramfs should not be used. If you compiled the needed drivers into the kernel then you will not need this ramdisk file to aid in booting. An initramfs is created the when make install is run.

- The following command updates the grub configuration file "/boot/grub/grub.cfg" to include a line for the new kernel. Using the 40_custom file in the the /etc/grub.d/ directory put an entry for your newly created kernel. A sample entry can be found in the existing /boot/grub/grub.cfg file. You may want to remove the quiet kernel command line parameter to see some of the boot messages. Run the following command to generate a new grub.cfg after you edit 40_custom.

  update-grub

- You may need to run grub-install to ensure that your partition (sda5) is loaded by grub on boot.
- Reboot the system, selecting your new kernel from the boot loader menu. Watch the messages. See if it works. If it does not, reboot with the old kernel, try to fix what went wrong, and repeat until you have a working new kernel.

Linux Kernel Module

Now, using the new kernel, compile and test the simple "*hello*" kernel module. Create a new directory, and download both the .c source file and Makefile into that directory from the class website "code" page:

http://ww2.cs.fsu.edu/~stanovic/teaching/ldd/code/assign1/Makefile
http://ww2.cs.fsu.edu/~stanovic/teaching/ldd/code/assign1/hello.c

Compile the module using make and test it, by executing the following commands:

```
insmod ./hello.ko
lsmod
rmmod hello
```

(Look at the end of /var/log/syslog or at the end of the output of the command "dmesg" to see any module output.)

- Now, make a copy of the module code and change it so that *module_init()* returns 1, recompile, and retest. What happens? Why?
- Now, change the module so that either *module_init()* or *module_exit()* does some bad thing (e.g., divide by zero). Create another file to do this, with a new name. Modify the **Makefile** to add your new module name to the list obj-m. Compile your module. If it compiles OK, test it, but ***before testing it***, use "sync" to synchronize the file system in-memory structures with those on disk, since the system may crash such that you will need to reboot.

Common Kernel Errors

- The following are some typical errors. Some of these may be caught at compilation or link time, and some may only cause problems when you call *insmod* with the module.
    - division by zero
    - dereferencing a null pointer
    - returning no value or a value other than zero from *init_mod*
    - calling a C library routine (*e.g., malloc()* or *printf()*) from inside a kernel module
    - performing a floating point operation
    - whatever else that you think might cause a problem
- Repeat a few such experiments, and see if you can hang or crash the entire system. Keep a journal of what you try, and what affects you observed. Save copies of the "bad" modules you tested, giving them different names, so that you can demonstrate them later.
- *Do not go overboard on this part of the assignment.* The objective is to expose you to the various ways a kernel failure can manifest itself, in preparation for testing your own code. The objective is not to do the most damage possible to your system. In particular, it would better if you you do not trash your hard drive, since reinstalling the entire system

takes a frustratingly long time. For example, a student once tried writing garbage into memory, walking downward through the kernel memory toward address zero. Some of that memory is mapped to hardware devices, like the disk controller, and some of it is used for buffers, like disk I/O buffers. Somehow, he managed to corrupt some of the OS files on the hard drive.

- Beware that errors in the kernel sometimes have no immediate visible effect, but they may have a delayed effect that is disastrous. This is generally the case if you write garbage into random locations of kernel memory. The location you corrupt may not be referenced to a while. It will be referenced later, and then the effect will occur. Therefore, you cannot assume that the thing you did most recently is necessarily the cause of a crash.
- As a consequence, during actual kernel development, one needs to test new kernel code extensively, letting the system run long enough (and with enough other activities going on) to give you confidence that the new code has no harmful side-effects.
- When you are debugging your own code, if you have tested some code that seems to have gone badly wrong, but not badly enough to crash the system (yet), you may want to reboot the system before testing your revised code. I have seen cases where a system crashed while executing the corrected code, but the crash was because of the delayed effect of damage that had been done by the previously executed (bad) version of the code. When this happens, the student sometimes mistakenly thinks the new code is bad, too, and becomes very confused.
- Prepare a copy of your journal, and have it ready to hand to the instructor when you demonstrate your project.