Laboratory Assignment #1

(Version 2.0 – typo fixes)

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 step through the mechanical steps of configuring, compiling, installing, and testing a kernel. You will then do the same for a trivial kernel module, given in the 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 a kernel crash.

Tasks:
Perform the following, on the machine in the lab that is assigned to you, or on a machine of your own that you can bring in to demonstrate. If you do not use one of the lab machines, take care to check that it has a parallel port, since you will be using the same machine for the first three assignments and for assignment #3 you will need the parallel port.

These instructions are for use with a particular version of the Linux kernel source tree (3.2.36) and a particular Linux system distribution (Ubuntu 12.04 LTS or Debian 6.0). If you are using a different Linux distribution you will need to adapt the instructions, or install a compatible distribution.

The detailed instructions 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 my description, please let me know.

Overview

If you are using a machines in the lab (LOV 016):

- Find a machine that no one else has yet claimed. The machines for this course are on the right side of the room, as you enter. The monitors should have tiny white labels with numbers in the range "0" .. "9" on them. Already-claimed machines should have notes on them (see below). The initial root password of each machine should be the one given to you in class. If you cannot log in

as "root" with this password it probably means that somebody else has claimed the machine but forgotten to put his/her name on it.
- Change the root password to one that you will remember. Then, using tape and paper or a Post-It, mark the monitor of the machine with your name. Leave room for the IP address and hostname, which you will find out later (see below).

If you are using your own machine, I recommend:
- Be prepared for the possibility that your machine's contents may be corrupted, requiring you to reload the entire system, possibly losing whatever else you had on the system.
- Consider setting up your system for dual-boot. This can allow you to run Linux in separate disk partitions from your normal OS installation, reducing (but not eliminating) the chance that you will corrupt your other system.
- Avoid partitioning your disk in a way that relies on using Logical Volume Manager (LVM).
- 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.
- Use a Linux distribution that has a kernel, no later than 3.2.x.

User Accounts Setup

Power on and boot up the machine, if it is not already running, and log in as "root". If you did not do so previously, create yourself at least one ordinary user account, so that you don't have to do all your work as "root". 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. To add a user account to the system perform the following steps:
- sudo adduser partner will create a new user named partner and setup the associated /home/partner directory structure
- sudo adduser partner sudo will add the user you created above (partner in this example) to the sudoers list. This, effectively, gives partner root access.

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 "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 options is to use sudo to run individual commands as root.

Minimal Services

If you sitting in front of the computer at the physical console, you can select any of six virtual consoles using the function keys F1-F6 + CTRL + ALT. F1 is the initial virtual console. F7 is reserved for the X-windows display. For example, if you are running X-windows, you can switch over to virtual console #2 using the F2 + CTRL + ALT key to log in as root and do something, then switch back to the X-windows display using F7 + CTRL + ALT.

Assuming you performed the default installation, the machine will probably come up with an X-windows display manager screen. If so, edit the file */etc/init/rc-sysinit.conf* to change the initial run-level from 2 to 1 edit the line to be *env DEFAULT_RUNLEVEL=1*. (You can get X-windows back later by changing the runlevel from 2 to 5, of by just executing the startx command from a login shell.) If the machine comes up with just a login prompt, skip this step.

Shut off all unnecessary system services. The objective here is to allow your machine to boot quickly, and to free up as much as possible of the memory so that your kernel compilations will go as quickly as possible.

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. **Do not power-cycle your system to reboot, if possible**. It is will prematurely age your hardware.

The Linux Kernel

Get a copy of the Linux kernel sources, 3.2.36. You can download them from **http://www.kernel.org**.

You may need to install additional deb (Debian package manager files) to compile and build a kernel. For Ubuntu 12.04 LTS or Debian 6.0, you need to install at least *libncurses5-dev*. Assuming you have networking up and running, you can do this using the apt-get and apt-cache utility. You might then want to try apt-get update, to get your list of packages up-to-date. Then execute apt-get upgrade to get your system up-to-date. To install individual packages, do sudo apt-get install libncurses5-dev, etc. You can read more about *apt-get* and *apt-cache* in the man-page.

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 xjpf linux-3.2.36.tar.bz2
sudo ln -s `pwd`/linux-3.2.36 /usr/src/linux
```

- If your file is gzipped instead of bzip2ed, you use the tar option "z" instead of "j".
- Once you have the kernel source tree uncompressed and untarred, cd into the source directory (*/usr/src/linux* in the examples above), 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.
- 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.
    - Open up the computer's case and read the labels on the components.
    - 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 backup 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 involves quite a bit of guesswork, 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.*
- Do the following *make* steps:

```
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. The third command is OS specific and will copy the new kernel into the directory "/boot".
- Next, you may need to create an initrd image in the /boot directory. If you compiled the needed drives into the kernel then you will not need this ramdisk file to aid in booting. If you are using Debian, you will probably want to create this ramdisk using the following command:

  ```
  mkinitramfs –o /boot/initrd.img-3.2.36 3.2.36
  ```

- Finally, the following command updates the Grub bootstrap loader configuration file "/boot/grub/grub.cfg" to include a line for the new kernel. The third command, make install, performs many operations behind the scenes. Examine the /etc/grub.d/ directory structure **before** and **after** you run the above commands to see the changes. Also look in the /boot/grub/grub.cfg file for your kernel entry.

  ```
  update-grub
  ```

- For extra credit remove the created initrd from the /boot/ directory as well as the references in **/etc/grub.d/***.
- 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 configuration script. If you run into this sort of problem, you are reduced to guesswork based on the compilation or linkage error messages. 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.
- 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. (If you want to learn more about Grub, you will find that you have options to temporarily modify menu lines on-line, to work around typos in the Grub configuration, but that is beyond the scope of these instructions.)
- When you reach a point of frustration -- from too many cycles of pruning, recompiling, testing, crashing, and rebooting -- stop. Take whatever kernel you have that will compile and run, and go on to the next step. If you are working on a machine in the lab and have not succeeded in producing *any* working kernel from the 3.2 source tree, ask the instructor for a copy of a working reduced configuration file, copy it into your Linux source directory under the name ".config", run "make oldconfig", and then "make; make modules_install; make". It should give you a modest-sized kernel that ran on at least one of the machines in the lab.

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/~diesburg/courses/dd/code/Makefile
http://ww2.cs.fsu.edu/~diesburg/courses/dd/code/hello.c

After you have compiled the module, to get *hello.o*, 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.
- You will need to modify the Makefile to add the name of your new module to it. For example, if you call your new module source file "hello1.c", you will need to add "hello1.o" to the line
- `obj-m := hello.o .... kdataalign.o`
- What happens? Why?
- Now, change the module so that either *module_init()* or *module_exit()* does some bad thing. 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 it is likely that the system will crash, and 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
  - 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.

Advice:

- Get started right away.
- *Feel free to ask for information/help from other students with the kernel configuration and similar technical set-up details, but don't let them do the work for you. Do it on your own. If you can't do this simple exercise by yourself now, you won't be able to do the other exercises later. In particular, do not ask someone else what changes they made to the kernel module, and what happened to them (until after the due date and all assignments are turned in). The odds of two people in the class having the same exact results are very slim, so if you just copy I am likely to notice.*

Delivery Method:

1. In class on the due date, arrange an appointment with the instructor 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.*
3. Please complete the quick Assignment 1 Survey on Blackboard. All survey answers are anonymous, but I can tell if you have completed the survey. This helps me improve the class.

Assessment:

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

1. Unnecessary system services shut off.
2. Kernel trimmed and unnecessary modules not compiled.
3. New kernel added to grub.
4. “hello” kernel module compiles in new kernel.
5. Bad modules created.
6. Assignment journal kept.

Extra credit will be given if the initrd image is removed.

Deductions will be made for:

1. Uncompleted steps.
2. Inability to explain what you have done.
3. A very large kernel:
    - Amount of memory in use, as shown by `/proc/meminfo`
    - Size of kernel image in `/boot/vmlinuz....`
    - Number of unnecessary modules loaded, shown by `lsmod` and by applying the `du` command to the subdirectory of `/lib/modules/...` for your kernel version.
4. An excessive set of services running, as shown by `ls -le`.
5. Having to edit code, recompile, etc. during demonstration of kernel failure modes.