Group Lab: Configuring and compiling the Linux kernel

Hands-On Unix System Administration DeCal

Lab 10 — 05 November 2012

Due 26 November at 6:10 PM for "extra credit"

1 Introduction

Recall that the kernel is essentially the core of an operating system. It is responsible for managing memory and processes, allocating and managing system resources, controlling and mediating access to hardware, among many other responsibilities.

In this lab we'll introduce you to the Linux kernel and briefly step through the kernel compilation process. You will be working with your group on your final project VM. You might want to take a snapshot of your VM before you begin the lab. This lab assumes that you already have a working VM set up (with an operating system and such). You and your group will turn in **one** lab, with all your names on it + your group name.

The goal of this lab is to give you a general idea of how to configure, compile, and boot from a new kernel. As scary as that might sound, it really isn't that bad. Now you can tell your friends that you know how to compile a linux kernel and they'll be in awe of your uber smartness.

Because we want you to focus on your final project, you won't be penalized if you are unable to boot from your new kernel when you finish the lab. Please document any difficulties you have along the way, any error messages you come across, and how you debugged them. Though you won't be penalized for some incompleteness (this is a sort of experimental lab we're trying), *please make an honest effort*, this is supposed to be fun!

2 Fetching the kernel source

To begin, let's fetch the kernel source from kernel.org with `wget -c` (I'm not imposing any version restrictions on you since this exercise is purely for learning purposes)

Say you want to fetch the latest stable kernel from kernel.org:

```
$ wget -c http://www.kernel.org/pub/linux/kernel/v3.0/linux-3.6.6.tar.bz2
```

Now copy this archive to your build directory (e.g., some directory in your homedir. in this example, I am copying the archive to a directory I made in my homedir called `kernelbuild`):

```
$ cp linux-3.6.6.tar.bz2 ~/kernelbuild/
```

Unpack the archive and `cd` into the source directory.

```
$ cd ~/kernelbuild
$ tar -xvjf linux-3.6.6.tar.bz2
$ cd linux-3.6.6
```

Run `ls` and explore for a bit. You'll see many directories containing C code organized by various functions such as memory management (in directory mm/), and other stuff like architecture-specific code (arch/), filesystems (fs/), drivers (drivers/). If you're interested in reading more about what are in these directories, here's a link to a webpage with pretty good explanations: `http://www.xml.com/ldd/chapter/book/ch16.html`

Next, run `make mrproper` to ensure that the source tree is clean.

3 Kernel build configuration

Now it's time to configure and customize your kernel.

You probably *don't* want to configure your kernel from scratch (as that takes a lot of time), so let's use the configuration from the currently running kernel. If you are compiling a kernel for the first time, it is usually a good idea to take the configuration of your existing (working!) kernel as a starting point for the configuration of your new kernel. Typically the current kernel configuration is saved in a file under /boot, in Debian-based distros (for example, something like /boot/config-2.6.3).

Copy this config file over to your build directory, and rename it `.config`:

```
$ cp /boot/config-blahblah ~/kernelbuild/linux-3.6.6/.config
```

If you don't find it there, you might have a config file in /proc (/proc/config.gz). In that case, use zcat to obtain its contents and copy it over to a `.config` file in your build directory.

```
$ zcat /proc/config.gz > ~/kernelbuild/linux-3.6.6/.config
```

Now that we have the old configuration in place run:

```
$ make oldconfig
```

This basically lets you start with a pre-existing .config file and prompts the user for options in the current kernel source that are not found in the file. This is useful when taking an existing configuration and moving it to a new kernel (like what we're doing here!). When you run `make oldconfig`, you'll be prompted with a bunch of new options that weren't in the old configuration. For now, we won't get into the details of these options, go ahead and just select the defaults for all of them (just hitting enter will automatically select the default option).

After `make oldconfig`, you could (optionally) run `make menuconfig` to visually see and fine-tune your options in a more user-friendly interface. `make menuconfig` will present you with a nice menu that allows you to set options and see what options have already been set. (you will need the `ncurses` library) You don't have to make any changes here.

Exercise 1 Now that you have a configuration file for the new kernel, list three options you find in `.config`. What are they set to? (y/n/m?) What do you think each option means (e.g., what feature does it enable/disable)? (You will probably have to search the internets)

4 Compilation and Installation

We've finished configuring our kernel, so now it's time to compile it. Run:

```
$ make
```

or

```
$ make -jN
```

to spawn multiple jobs, where N is usually the number of cores + 1. This will speed things up.

Compilation time can vary from as short as 15 minutes to well over an hour. Speed is mostly based on how many options/modules were selected, as well as processor power. Since you're most likely using a generic configuration, many modules will be compiled and this overall process can take some time. Go grab a cup of coffee or something while you wait.

Exercise 2 How long did compilation take? Did you use multiple threads? How many? (make -jN) Did you run into problems during the process?

After the kernel has finished compiling, we need to install the kernel **modules**.

Modules are pieces of code that can be loaded and unloaded into the kernel on demand. They extend the functionality of the kernel without requiring you to reboot your machine. For instance, one type of module is the device driver, which allows the kernel to interact with hardware connected to the system.

Exercise 3 Run `lsmod` to get a list of currently loaded kernel modules on your system. List three of them, and describe their functions (may need to search the internet for hints!)

Now, back to installing modules. Run, as root, in your build directory:

```
# make modules_install
```

This copies the compiled modules into /lib/modules/[kernel version]. They are installed in a specific directory so that modules can be kept separate from those used by other kernels on your machine.

Finally, we will install the kernel image we just built to the `/boot` directory. Later we will be configuring your bootloader to add an option to boot from this kernel.

IMPORTANT. Make sure you keep your existing kernel (in /boot) around as a backup. (check out /boot and see what sort of stuff's in there. Kernels are usually prefixed with vmlinux-

Finally, from your build directory, run:

```
# make install
```

5 Make initial RAM disk

The initial RAM disk (specified as the `initrd` option in the GRUB menu, or, the file "initramfs-YourKernelName.img in /boot") is an initial root file system that is mounted before the real root file system becomes available. I won't get into the details, but the initrd, in a nutshell, is a transient filesystem bound to the kernel and loaded as part of the kernel boot procedure. The initrd contains various executables and drivers that permit the real root file system to be mounted. Its lifetime is short, only serving as a bridge to the real root file system.

Let's build the ramdisk. The -k parameter accepts the kernel version, or the path to a kernel image:

```
# cd /boot; mkinitramfs -k 3.6.6 -o initrd.img-3.6.6
```

6 Configure the bootloader

A bootloader is basically a small program that loads the operating system into the computerwhen the system is booted and also starts the operating system.

We're now going to add an entry for your awesome new kernel in your bootloader's configuration file. (MOST likely, your bootloader is GRUB or GRUB2...doubt anyone will be using LILO, in that case, Google is your friend.), the following are instructions for GRUB Legacy.

If you're using GRUB Legacy (not GRUB2), you could probably run a command named `update-grub` that wil automatically update your grub menu file and add a new kernel entry for you. `update-grub` is a program used to generate the menu.lst file used by the grub bootloader. It works by looking in /boot for all files which start with "vmlinuz-". They will be treated as kernels, and grub menu entries will be created for each.

Try running the following to see what version of grub you have:

```
$ grub --version
```

Then,

```
# update-grub
```

or if you have GRUB2:

```
# update-grub2
```

If you don't have `update-grub` available, we can always edit /boot/grub/menu.lst manually and add an entry for your new kernel. It's easiest if you just copy an existing entry and modify the fields as appropriate. Here's a sample entry might look like:

```
title Arch Linux 3.6.6
root   (hd0,0)
kernel /vmlinuz-3.6.6 root=<root partition> ro
initrd /initramfs.img-3.6.6
```

The root partition simply refers to where the root directory (/) is stored. You'll most likely have an entry with the `root=` field already specified, so just copy that. Save the file. You should now have the following things in the /boot directory:

vmlinuz-YourKernelName (Kernel)

initramfs-YourKernelName.img (Ramdisk)

You're pretty much done. Take a deep breath and reboot your VM. Hopefully you'll see your new kernel in the bootloader menu, select it and pray that your machine boots!

If thinks appear borked, that's okay, you can still boot from the other kernel that you should have (select that one instead from the bootloader menu). Note any error messages you come across and list them. (for example, "I got dropped to a recovery shell, was unable to find root disk.")

If things miraculously worked, your machine should be running the new kernel. Congrats! Note the output of `uname -a` to verify things worked.

Exercise 4 Describe any difficulties you went through, any error messages you saw, etc. Did your machine successfully boot from your new kernel? Did things break? Was this a difficult lab? (This is more of a reflection exercise.)