Linux Kernel Compilation

Jason Kaufman

October 29, 2001

Background

The kernel is literally the core of a Linux system – it controls I/O, manages memory and processes, and so forth. One could easily write a very long and detailed account of exactly what the kernel does and how it does it, but I leave those tasks to more knowledgable authors. See the References for example texts. The focus of this discussion is purely the compilation of the kernel. It is assumed that the reader has a 2.4.x kernel installed.

Why Compile a Kernel?

Well, for several reasons. One could be that there is a critical error with a kernel you're currently using (read linux-2.4.11). Another may be that you bought some new hardware that needs a driver or two that your current kernel does not include, or is/are outdated. Finally, you could be developing new pieces of the kernel for device drivers, packet filtering, etc.

No matter what the reason is, the average user will need or want to upgrade the kernel at some point during the life of a Linux system.

1 Getting the Source

The kernel source can be found at the Linux Kernel Repository (`http://kernel.org/`), but you'll probably want to grab it from a mirror. With a quick look at the `/pub/linux/kernel/v2.4/` directory, you'll see various files: `{linux, patch}*.gz`, `{linux, patch}*.bz2`, `*.sign`, etc. The `linux*` refers to a package which contains the entire kernel source. It does not matter if you get the gzip'ed or bzip2'ed archive, they both give you the same source. However, make sure to get the correct `*.sign` file, as it contains a PGP signature for a specific archive.

For example you might download:

```
linux-2.4.13.tar.bz2
linux-2.4.13.tar.bz2.sign
```

or:

```
patch-2.4.13.bz2
patch-2.4.13.bz2.sign
```

Notice in the latter example, a "patch" was selected. Patches are useful if you already have an older kernel source tree unpacked. When you apply the patch using `patch -p1 <patch-2.4.*`, the old kernel source will be upgraded to whatever version the patch is. Patches also conserve bandwidth, and take much less time to implement.

In either the case of downloading a 20MB+ source tarball or a 1MB patch, you will need to check the PGP signature of the kernel archive. This is very important, and it's probably something that most users don't do. Knowing that the signature on the archive was good tells you that one – the file you downloaded actually came from a good source and was not placed there by someone else – and two – the archive was not corrupted in transit.

For those PPC users out there, some tweaking to the kernel source is usually required to get something that's operational. Luckily, there's people who have already done a bulk of the work for you. The best place to go for tweaked kernel source and binaries is `http://ppckernel.org`; a good kernel maintainer is Benjamin Herrenschmidt, otherwise known as BenH. His kernels tend to have better USB support, and now he has VGA-out support for PowerBooks.

2 Unpacking and Compiling

Eventually, you will have your source in one form or another and be ready to fully unpack the source into your home directory. This is also very important and should be said again; unpack the source into your home directory. You DO NOT want to extract it into `/usr/src/`. This may overwrite the original kernel headers that were installed with your system that other critical packages rely on, e.g. glibc. Official Linux kernel tarballs always extract into a `linux/` directory. It is always a good thing to give tar the `-tvf` options to see where it will extract files from the archive.

One thing I like to do at this point is make a backup the list of all the hardware that the current kernel has found in my system. Do this with `dmesg > hw_list.txt`. Now you can look through that file and see exactly which drivers the current kernel had to load. This helps out a lot when you're selecting kernel options the first time. If you've already compiled a kernel here before, another good idea is to copy the backup `.config` from your old build to the new source tree. This way, you will only be prompted for or have to select a few new kernel options later, rather than going through all the options by hand again.

On to the kernel configuration process. First, go into the top level of the kernel source tree and type `make mrproper` if you don't want to use an old kernel `.config` file. `mrproper` cleans the source tree and also removes any configuration files you may have created in the tree.

Now there's four main options for configuring the kernel. You can `make` a `config`, `oldconfig`, `menuconfig`, or `xconfig`. The first option is done within the shell, and asks whether or not you want to include each option in the kernel step by step. Although it doesn't take long to go through, good luck going back and correcting a mistake in the same run – it doesn't let you go back unless you restart the script. `make oldconfig` is quick and dirty, it simply takes `.config`, parses that as your configure options for the kernel, and creates a `.config.old`. `make menuconfig` is an ncurses-based configuration script. And last, but not least is `make xconfig`; it's an X-based script that allows you to both save and load in configuration files.

Select your options. I can't really say anything too specific here because everyone's machine is different. A couple pieces of advice though: if you have an IDE CD burner, don't enable ATAPI CDROM support. In this case, the kernel needs to see the burner as a SCSI drive and disabling this option prevents it from ever loading the IDE driver. You need generic SCSI, SCSI CDROM, and SCSI emulation support. Next, if you have a NIC, don't forget to enable UNIX domain sockets support. This is usually enabled by default, but I once spent about an hour trying to figure out why I couldn't browse the web because I left this option out.

Most kernel options can be built as "modules". Kernel modules are just that – object files which can be placed into or taken out of the kernel at any time without rebooting. They are handy for kernel development and for testing drivers for hardware that you may be unsure about. Linus Torvalds doesn't use modules, so I generally don't see a need to either. I only ever build sound support as a module, just so `init` doesn't complain about it on boot and confuse users.

Now you're ready to compile. Just type `make dep clean bzImage` and watch all the compilation that's going on. If anything fails, take a good look at the error. If what the compile process was working on is not mission-critical, take it out for now by running one of the configure scripts again. If things go all right and the compilation ends without any fatal errors, you now have built a compressed kernel image. In case you built modules, you only need to run `make modules` and then as root, `make modules_install`.

3 Installing the New Kernel

3.1 x86

From now on, you'll have to be root. What follows are specific instructions for x86.

```
cp arch/i386/boot/bzImage /boot/vmlinuz-2.4.x
cp System.map /boot/System.map-2.4.x
```

Assuming you're using LILO, open up `/etc/lilo.conf` in the editor of your choice. Most likely one already exists and has information about your old kernels and other devices that can be used to boot your system.

Find an old entry like this one:

```
image=/boot/vmlinuz-2.4.9
        label=linux-249
        root=/dev/hda1
```

Copy it and replace the old kernel version numbers with the new ones. Don't set the default image to boot as your new one yet – test it out to make sure it works. Save the file and run `/sbin/lilo`. Doing this tells LILO about your new kernel; if you don't you won't be able to use your new kernel until you do. Reboot the system and enjoy your new kernel. Afterwards, you may want to change the default kernel in LILO to your new one. Another good thing to do is to make a bootdisk with `mkbootdisk`, but I leave it as an exercise for the reader.

3.2 PPC

Kernel installation on PPC varies from machine to machine. New World machines tend to use Yaboot or ybin. BootX recently merged with Yaboot, so if you're not using Yaboot now might be a good time to start. Typically, PPC's need a small HFS partition where the bootloader and uncompressed kernel images sit. Some suggest that this only needs to be 10MB, and that should be enough depending on how many kernels are sitting on the partition. I have mine labeled as `/boot`, but Linux does not actually use this partition as an x86 would during boot. Copy your files over to `boot` as root:

```
cp arch/ppc/boot/images/vmlinux.gz /boot/vmlinuz-2.4.x
cp System.map /boot/System.map-2.4.x
```

Now mount the HFS boot partition like this:

```
mount -t hfs /dev/hdaXX /mnt/temp_boot/
```

Copy an uncompressed kernel to the HFS partition:

```
gunzip arch/ppc/boot/images/vmlinux.gz
cp arch/ppc/boot/images/vmlinux /mnt/temp_boot/Linux\ Kernels/vmlinux-2.4.x
```

Now make an entry in `/mnt/temp_boot/System\ Folder/yaboot.conf` like this:

```
label = 2.4.12
image = vmlinux-2.4.12
root = /dev/hda13
append = "video=aty128fb"
```

No other configuration is needed. Unmount the HFS partition, reboot, and pray that it doesn't destroy your system. Hehehe...

Notes

4 References

Websites

Linux Kernel HOWTO
`http://linuxdoc.org/HOWTO/Kernel-HOWTO.html`
Linux Kernel Repository
`http://kernel.org/`
Linux Kernel OpenPGP Signature Page
`http://kernel.org/signature.html` Overview of PPC Kernel Maintainers
`http://ppckernel.org`
BenH's Site
`http://master.penguinppc.org/~benh/`

Books

Understanding the Linux Kernel, ISBN: 0596000022