Compiling the Linux kernel

Section 1. Tutorial tips

Should I take this tutorial?

Do I need experience?

If you're relatively new to Linux, or any other UNIX or UNIX-like operating system, and would like to learn how to configure, compile, and install the Linux kernel, then this tutorial is for you. In it, you'll learn what the Linux kernel is, what modules are, and how to download, compile, and install a new kernel. This tutorial will walk you through the complete kernel compilation process.

If you have previously compiled a Linux kernel yourself, you may find this tutorial to be a good refresher course.

Navigation

Navigating through the tutorial is easy:

1. Use the Next and Previous buttons to move forward and backward through the tutorial.
2. Use the Menu button to return to the tutorial menu.
3. If you'd like to tell us what you think, use the Feedback button.
4. If you have a question for the author about the content of the tutorial, use the Contact button.

Contact

For technical questions about the content of this tutorial, contact the author, Daniel Robbins, at drobbins@gentoo.org.

Daniel resides in Albuquerque, New Mexico. He is the President/CEO of Gentoo Technologies, Inc., the Chief Architect of the *Gentoo Project* and a contributing author of several books published by MacMillan: Caldera OpenLinux Unleashed, SuSE Linux Unleashed, and Samba Unleashed. Daniel has been involved with computers in some fashion since the second grade, when he was first exposed to the Logo programming language as well as a potentially dangerous dose of Pac Man. This probably explains why he has since served as a Lead Graphic Artist at SONY Electronic Publishing/Psygnosis. Daniel enjoys spending time with his wife, Mary, and his new baby daughter, Hadassah.

Section 2. Introducing the kernel

The kernel is...Linux!

What do you think of when you hear the word "Linux"? When I hear it, I typically think of an entire Linux distribution and all the cooperating programs that make the distribution work.

However, you may be surprised to find out that, technically, Linux is a kernel, and a kernel only. While the other parts of what we commonly call "Linux" (such as a shell and compiler) are essential parts of a distribution, they are technically separate from Linux (the kernel). While many people use the word "Linux" to mean "Linux-based distribution," everyone can at least agree that the Linux kernel is the *heart* of every distribution.

Interfacing with hardware

The primary role of the Linux kernel is to interface directly with the hardware in your system. The kernel provides a *layer of abstraction* between the raw hardware and application programs. This way, the programs themselves do not need to know the details of your specific motherboard chipset or disk controller -- they can instead operate at the higher level of reading and writing files to disk, for example.

CPU abstraction

The Linux kernel also provides a level of abstraction on top of the processor(s) in your system -- allowing for multiple programs to appear to run simultaneously. Linux does this by allowing several UNIX *processes* to run at once -- and the kernel takes care of giving each one a fair share of the processor(s).

A Linux kernel can support either a single or multiple CPUs -- and the kernel that you are using now is either uniprocessor-aware (UP-aware) or symmetric multiprocessor-aware (SMP-aware). If you happen to have an SMP motherboard, but you're using a UP kernel, Linux won't "see" your extra processors! To fix this, you'll want to compile a special SMP kernel for your hardware. Currently, SMP kernels will also work on uniprocessor systems, but at a slight performance hit.

Abstracting I/O

The kernel also handles the much-needed task of abstracting all forms of file I/O. Imagine what would happen if every program had to interface with your particular hardware directly -- if you changed disk controllers, all your programs would stop working! Fortunately, the Linux kernel follows the UNIX model of providing a simple abstraction of disk I/O that all programs can use. That way, your favorite database doesn't need to be concerned whether it is storing data on an IDE disk, a SCSI RAID array, or a network-mounted file system.

Networking Central

One of Linux's main claims to fame is its robust networking, especially TCP/IP support. And, if you guessed that the TCP/IP stack is in the Linux kernel, you're right! The kernel provides a nice, high-level interface for programs that want to send data over the network. Behind the scenes, the Linux kernel interfaces directly with your particular ethernet card or modem, and handles the low-level Internet communication details.

Networking goodies

One of the greatest things about Linux is all of the useful features that are available in the kernel, especially those related to networking. For example, you can configure a kernel that will allow your entire home network to access the Internet via your Linux modem -- this is called IP Masquerading, or IP NAT.

Additionally, the Linux kernel can be configured to export or mount network-based NFS file systems, allowing for other UNIX machines on your LAN to easily share data with your Linux system.

Booting, part 1

When you turn on your Linux-based system, the kernel is loaded from disk to memory by a boot loader, such as LILO. At this point, the kernel takes control of your system. The first thing it does is detect and initialize all the hardware that it finds -- and it has been compiled to support. Once the hardware has been initialized properly, it is then ready to run processes. The first process it runs is called "init", which is located in /sbin. Then, "init" starts additional processes, as specified in /etc/inittab.

Booting, part 2

"init" typically starts several copies of a program called "getty," which waits for logins from the console. After getty successfully processes a login request, your default shell is loaded (which is typically bash). Once you're in bash, you have the power to launch any program you'd like.

While all these new processes are started, the kernel is still in control, carefully time-slicing the CPU so that each process has a fair share. In addition, the kernel continues to provide hardware abstraction and networking services for the various running processes.

Introducing...modules!

All recent Linux kernels support kernel modules. Kernel modules are really neat things -- they're pieces of the kernel that reside on disk, until needed. As soon as the kernel needs the functionality of a particular module, it's loaded from disk, automatically integrated with the kernel, and available for use. In addition, if a kernel module hasn't been used for several minutes, the kernel can voluntarily unload it from memory -- something that's called "autocleaning."

Modules, part deux

Kernel modules live in /lib/modules, and each module has a ".o" at the end of its name. As you may guess, modules each represent a particular component of kernel functionality -- one module may provide FAT filesystem support, while another may support a particular ISA ethernet card.

Modules allow you to have a low kernel memory footprint. You can create a kernel that contains only the features necessary for booting your computer, and all other features can be loaded from modules on demand. Because the kernel autocleans any modules it loads, your system's memory can be put to good use.

Modules -- important stuff!

You can't put *everything* in a module. Because modules are stored on disk, your bootable kernel image needs to have compiled-in support for your disk controller as well as for your native file system (typically the ext2 filesystem). If you don't have these essential components compiled into your kernel image (but compile them as modules instead), then your kernel won't have the necessary ability to load these modules from disk -- creating a rather ugly chicken-and-egg problem!

Progress quiz

It's time for a quick quiz. True or false: It's not a good idea to put your primary disk controller driver in a loadable module.

A. True
B. False

(The correct answer is "A. True")

Section 3. Locating and downloading sources

Kernel versions

To compile a recent kernel, you need to download the sources first. But before you download the kernel sources, you need to know what you're looking for. The first question to ask yourself is this -- do you want to use a stable or *experimental* kernel?

Stable kernels always have an even second digit -- for example, 2.0.38, 2.2.15, 2.2.18, and 2.4.1 are all considered "stable" kernels (due to the 0, 2, 2, and 4, respectively.) If you'd like to test out an experimental kernel, you'll typically look for the highest-numbered kernel that has an odd second number. For example, 2.3.99 and 2.1.38 are both experimental kernels (due to their 3 and 1, respectively).

Kernel version history

The 2.2 series is considered a modern, stable kernel. If "modern" and "stable" are things that sound good to you, look for a 2.2 kernel with the highest third number you can find (2.2.16 is the most recent version at the moment).

While the 2.2 series kernel was being developed, the 2.3 series began. This series was created to serve as a testing ground for new, advanced features that would eventually show up in the stable 2.4 series. As of right now, the 2.3 series has already reached 2.3.99, and 2.3 development has stopped. These days, developers are working on getting the 2.4.0 *test* kernels into shape. If you'd like to be on the cutting-edge, you'll want to try the most recent 2.4.0-test kernel you can get your hands on.

2.4 kernel warning

Once a *real* 2.4 series kernel comes out (like 2.4.0), don't assume that the kernel is ready for use on a mission-critical system like a server. Even though 2.4 is supposed to be a stable series, early 2.4 kernels are likely to be not quite up to snuff. As is often the case in the computer industry, the first version of anything can have fairly sizable bugs. While this may not be a problem if you're testing the kernel on your home workstation, it is a risk you may want to avoid when you machine provides valuable services to others.

Downloading the kernel

If you simply want to compile a new version of your installed kernel (for example, to enable SMP support), then no downloading is necessary -- skip past this and the next panel.

You can find kernels at *http://www.kernel.org/pub/linux/kernel*. When you go there, you'll find the kernel sources organized into several different directories, based on kernel version (v2.2, v2.3, etc.) Inside each directory, you'll find files labelled "linux-x.y.z.tar.gz" and "linux-x.y.z.tar.bz2". These are the Linux kernel sources. You'll also see files labelled "patch-x.y.z.gz" and "patch-x.y.z.bz2". These files are patches that can be used to update the previous version of complete kernel sources. If you want to compile a new kernel release, you'll need to download one of the "linux" files.

Unpacking the kernel

If you downloaded a new kernel from kernel.org, now it's time to unpack it. To do so, cd into /usr/src. If there is an existing "linux" directory there, move it to "linux.old" ("mv linux linux.old", as root.)

Now, it's time to extract the new kernel. While still in /usr/src, type `tar xzvf /path/to/my/kernel-x.y.z.tar.gz` or `cat /path/to/my/kernel-x.y.z.tar.bz2 | bzip2 -d | tar xvf -`, depending on whether your sources are compressed with gzip or bzip2. After typing this, your new kernel sources will be extracted into a new "linux" directory. Beware -- the full kernel sources typically occupy more than 50 megabytes on disk!

Section 4. Configuring the kernel

Let's talk configuration

Before you compile your kernel, you need to configure it. Configuration is your opportunity to control exactly what kernel features are enabled (and disabled) in your new kernel. You'll also be in control of what parts get compiled into the kernel binary image (which gets loaded at boot-time), and what parts get compiled into load-on-demand kernel module files.

The old-fashioned way of configuring a kernel was a tremendous pain, and involved entering /usr/src/linux and typing `make config`. While make config still works, please don't try to use this method to configure your kernel -- unless you like answering hundreds (yes, hundreds!) of yes/no questions on the command line.

The New Way to configure

We more modern folks, instead of typing `make config`, type either `make menuconfig` or `make xconfig`. If you'd like to configure your kernel, type one of these options. If you type `make menuconfig`, you'll get a nice text-based color menu system that you can use to configure the kernel. If you type `make xconfig`, you'll get a very nice X-based GUI that can be used to configure various kernel options. Here's a screenshot of "make menuconfig":

When using "make menuconfig", options that have a "< >" to their left can be compiled as a module. When the option is highlighted, hit the space bar to toggle whether the option is deselected ("< >"), selected to be compiled into the kernel image ("<*>") or selected to be compiled as a module ("<M>").

Configuration tips

There are tons of kernel options, and there's no room to explain them all here -- so please take advantage of the kernel's built-in help functionality. Almost every option is described in at least some detail, and each one normally contains the line "If you don't know what this means, type Y (or N)." These hints keep you out of trouble if you happen to have no clue what a particular option actually does. To access help, highlight the option you have a question about and press the "?" key.

Section 5. Compiling and installing the kernel

make dep; make clean

Once your kernel is configured, it's time to get it compiled. Before we can compile it, we need to generate dependency information and also clean out any old "compiled stuff." This can be accomplished by typing: `make dep; make clean` while in /usr/src/linux.

---

make bzImage

Now, it's time to compile the actual binary kernel image. Type `make bzImage`. After several minutes, compilation will complete and you'll find the bzImage file in /usr/src/linux/arch/i386/boot (for an x86 PC kernel). We'll show you how to install the new kernel image in a bit, but now it's time for the modules.

Compiling modules

Now that the bzImage is done, it's time to compile the modules. Even if you didn't enable any modules when you configured the kernel, don't skip this step -- it's good to get into the habit of compiling modules immediately after a bzImage. And, if you really have no modules enabled for compilation, this step will go really quickly. Type `make modules; make modules_install`. This will cause the modules to be compiled and then installed into /usr/lib/<kernelversion>.

Congratulations! Your kernel is now fully compiled, and your modules are all compiled and installed. Now, it's time to reconfigure LILO so that you can boot the new kernel.

Progress quiz

Let's see how well you were paying attention :) True or false: for compilation, "make dep" is optional.

A. True
B. False

(The correct answer is "B. False")

Section 6. Boot configuration

LILO introduction

It's finally time to reconfigure LILO so that it loads the new kernel. LILO is the most popular Linux boot loader, and is used by all popular Linux distributions. The first thing you'll want to do is take a look at your /etc/lilo.conf file. It will contain a line that says something like "image=/vmlinuz" This line tells LILO where it should look for the kernel.

Boot configuration, part 2

To configure LILO to boot the new kernel, you have two options. The first is to overwrite your existing kernel -- this is risky unless you have some kind of emergency boot method, such a boot disk with this particular kernel on it.

The safer option is to configure LILO so that it can boot either the new or the old kernel. LILO can be configured to boot the new kernel by default, but still provide a way for you to select your older kernel if you happen to run into problems. This is the recommended option, and the one we'll show you how to perform.

Boot configuration, part 3

Your lilo.conf may look like this:

boot=/dev/hda delay=20 vga=normal root=/dev/hda1 read-only image=/vmlinuz label=linux

To add a new boot entry to your lilo.conf, do the following. First, copy /usr/src/linux/arch/i386/boot/bzImage to a file on your root partition, such as /vmlinuz2. Once it's there, duplicate the last three lines of your lilo.conf and add them again to the end of the file... we're almost there...

Boot configuration, part 4

Now, your lilo.conf should look like this:

boot=/dev/hda delay=20 vga=normal root=/dev/hda1 read-only image=/vmlinuz label=linux image=/vmlinuz label=linux

Now, change the first "image=" line to read "image=/vmlinuz2". Next, change the *second* "label=" line to read "label=oldlinux". Also, make sure there is a "delay=20" line near the top of the file -- if not, add one. If there is, make sure the number is at least twenty.

Boot configuration, part 5

Your *final* lilo.conf file will look something like this:

```
boot=/dev/hda delay=20 vga=normal root=/dev/hda1 read-only  image=/vmlinuz2
label=linux image=/vmlinuz label=oldlinux
```

After doing all this, you'll need to run "lilo" as root. This is very important! If you don't do this, the booting process won't work. Running "lilo" will give it an opportunity to update its boot map.

Boot configuration, an explanation

Now for an explanation of our changes. This lilo.conf file was set up to allow you to boot two different kernels. It'll allow you to boot your original kernel, located at /vmlinuz. It'll also allow you to boot your new kernel, located at /vmlinuz2. By default, it will try to boot your new kernel (because the image/label lines for the new kernel appear first in the configuration file).

If, for some reason, you need to boot the old kernel, simply reboot your computer and hold down the Shift key. LILO will detect this, and allow you to type in the label of the image you'd like to boot. To boot your old kernel, you'd type "oldlinux", and hit enter. To see a list of possible labels, you'd hit TAB.

Resources

Congratulations on compiling your own kernel! I hope everything went well. Here are some related resources where you can learn more about kernel compilation:

1. The Linux Kernel HOWTO, another good resource for kernel compilation instructions
2. The LILO, Linux Crash Rescue HOW-TO, which shows you how to create an emergency Linux boot disk
3. www.kernel.org, the site that hosts the Linux Kernel archives

Section 7. Wrapup

Your feedback

Please let us know whether this tutorial was helpful to you and how we could make it better. We'd also like to hear about other tutorial topics you'd like to see covered. Thanks!