

How To Compile and Install a 2.6.x Series Linux Kernel

Josh VanderLinden - 9 Feb 2007

The Linux kernel is the core component in any Linux distribution. Without a kernel, your computer would be essentially useless. It is the piece of software which allows interaction between you, your computer's applications, and your computer's hardware. With such a powerful role in your computing experience, it is important to keep your kernel up-to-date. Each new release provides more hardware support and many performance enhancements. It is also important to keep your kernel up-to-date for security purposes.

Let's upgrade our Linux kernels together. I will walk you through each of the steps I take, from beginning to end, to upgrade my kernel. Just as a warning, I prefer to do the whole process on the command line, so you might want to pull up a terminal, konsole, xterm or whatever you prefer to use for your command line operations.

First you need to download the kernel source code. Many Linux distributions provide specialized editions of the Linux kernel. Typically, you don't want to manually compile and install a custom kernel for these distributions. This does not mean that you can't, it simply means that you might be better off using the "official" kernels for your distribution, which can usually be obtained through your distribution's package manager. You can get the official, 100% free, and complete Linux kernel source code from <http://www.kernel.org/>. Look for "The latest stable version of the Linux kernel is:" and click the link on the F on the same line. Currently, the latest stable version is 2.6.20, and that's what I'll be using for this tutorial. Please note that commands which begin with a dollar sign (\$) are executed as a regular user and commands beginning with a pound sign (#) are executed as a superuser.

```
$ cd /home/user/download
$ wget http://www.kernel.org/pub/linux/kernel/v2.6/linux-2.6.20.tar.bz2
```

Now login as the superuser, and navigate to the `/usr/src` directory. Then extract the kernel source into that directory.

```
$ su -
# cd /usr/src
# tar jxf /home/user/download/linux-2.6.20.tar.bz2
```

You probably already have a symlink or shortcut called `linux` which points to your most recent kernel. If you do, delete the link and create another link to the new source tree. Then go into your kernel source tree.

```
# rm /usr/src/linux
# ln -s /usr/src/linux-2.6.20 /usr/src/linux
# cd /usr/src/linux
```

I like to identify each compile of my kernel uniquely, to make sure that I'm using the right one. To do that, you have to modify your `Makefile`

```
# vi Makefile
```

You will see the following lines, or something similar, at the very top of the file:

```
VERSION = 2
PATCHLEVEL = 6
SUBLEVEL = 20
EXTRAVERSION =
NAME = Homicidal Dwarf Hamster
```

Change the `EXTRAVERSION` property to something you want to use to identify this kernel. I will use ``-jcv1``

```
EXTRAVERSION = -jcv1
```

The rest of the `Makefile` should be fine. In fact, I discourage editing `Makefiles` unless you know what you're doing. This next step is totally optional, but I like to do it to save some time. You can copy your existing kernel's configuration file in order to have a very similar kernel configuration. My previous kernel version was 2.6.19.1, so this is the command I use:

```
# cp /usr/src/linux-2.6.19.1/.config /usr/src/linux/
```

Then I run ``make oldconfig`` or ``make silentoldconfig`` to update my older kernel configuration file to be able to handle newer features. If you use ``oldconfig`` you are required to specify whether or not you want the new features included in your kernel, whereas ``silentoldconfig`` will use the defaults determined by kernel developers (they usually know best), asking for minimal input. Let's update our configuration file and then customize it by running ``make menuconfig`` (there are several options here, such as ``make xconfig`` and ``make gconfig``, but I prefer the text-based `menuconfig`; there is another you can run by using ``make config``, which runs through each and every option available—it's scary).

```
# make silentoldconfig
# make menuconfig
```

`menuconfig` is a graphical command line application which lets you navigate the features offered by the kernel. Each computer is considerably different from the next, so it really does no good to provide a list of things that I tweak. However, it is important to note what some of the symbols are in

the `menuconfig` utility:

- M = Module. Modules are loaded when they are required and can contribute to the speed of your system
- * = built into the kernel. These are typically things which are necessary for your machine to function properly, such as support for your root file system.
- X = exclusively selected. You'll see this when you select what type of processor you have, for example.

One thing to note before we go further is *MAKE SURE YOUR KERNEL HAS BUILT-IN SUPPORT FOR YOUR ROOT FILE SYSTEM!!!!* My root file system is reiserfs. In my configuration, I made sure that reiserfs was marked with a star. If you don't do this, your kernel won't boot and you will be very frustrated. Trust me.

Your computer is probably quite different than mine, so you might want to just poke around and see if you recognize things that deal with your computer's hardware. Once you are done tweaking your kernel configuration, exit the configuration utility and make sure the configuration is stored in `/usr/src/linux/.config`

Next we get to build and install the kernel. After that, we have to add an entry to our boot manager so that we can try out our new kernel. The compilation part usually takes just about a half hour on my 2.2Ghz Turion64 processor with 1.25GB of RAM. It takes about 6 hours on my 300Mhz Pentium 2 with 32MB of RAM. Let's find out how long it takes for you to compile your kernel!

```
# time make
...
real    27m29.663s
user    23m34.476s
sys     2m56.575s
```

Now let's install the modules and install the appropriate files in the boot area:

```
# make modules_install
# make install
```

This is the part that always used to mess me up. I use Slackware Linux, which is more UNIX-ish than most distributions. It's actually the oldest surviving Linux distribution to date, but that's another story. For some reason, the ``make install`` command doesn't always work with Slackware. There is a process I use to setup my boot directory when I compile a new kernel. I wrote a simple shell script called ``fixkernelinstall`` to take care of it for me:

```

#!/bin/bash
# Configure my computer for a new kernel
# Author: Josh VanderLinden
# Assisted By: Dan Purcell

# if the user didn't supply a kernel number, ask for it
if [ $# -eq 0 ]; then
    echo -n "Kernel: "
    read kernel
else
    kernel=$1
fi

# determine root partition
echo "Determining root partition..."
rootpart=`mount -l | grep ' / ' | cut -f 1 -d\ `
echo "Root partition is $rootpart"

# copy kernel configuration file
cp /usr/src/linux/.config ./config-$kernel

# now rename everything
echo "Renaming files..."
mv System.map System.map-$kernel
mv vmlinuz vmlinuz-$kernel

# if the config file exists and it's a symlink, remove it
if [ -f 'config' -a `stat config | grep -c 'symbolic link'` =
'1' ]; then
    echo "Removing link to configuration file"
    rm config
else
    # otherwise it might be important
    echo "Renaming configuration file"
    mv config config.bak
fi

# Link files
echo "Creating symlinks..."
ln -s System.map-$kernel System.map
ln -s config-$kernel config
ln -s vmlinuz-$kernel vmlinuz

# Update lilo
echo "Adding entry to /etc/lilo.conf for $kernel"
echo "image = /boot/vmlinuz-$kernel" >> /etc/lilo.conf
echo " root = $rootpart" >> /etc/lilo.conf
echo " label = $kernel" >> /etc/lilo.conf
echo " read-only" >> /etc/lilo.conf

echo "Linux kernel $kernel has been configured."
echo "Please check your lilo configuration and run lilo before
rebooting"

```

I'm not an expert on shell scripts, so please feel free to offer suggestions for doing things better if you know how. This script uses the kernel version (given by the user) to setup by `/boot` directory properly. In my case, I run the script as

```
# cd /boot
# fixkernelinstall 2.6.20-jcv1
```

And the output is something like:

```
Determining root partition...
Root partition is /dev/hda5
Renaming files...
Renaming configuration file
Creating symlinks...
Adding entry to /etc/lilo.conf for 2.6.20-jcv1
Linux kernel 2.6.20-jcv1 has been configured.
Please check your lilo configuration and run lilo before rebooting
```

As you can see from the script, I use LILO instead of the arguably more popular GRUB. Either one works for me, but LILO is sufficient for my needs. If you want to use the same kind of script for a GRUB installation, just change the LILO part at the end to something like:

```
echo 'Adding entry to /boot/grub/menu.lst for $kernel'
echo '  title Linux on ($rootpart)' >> /boot/grub/menu.lst
echo '  root (hd0,4)' >> /boot/grub/menu.lst
echo '  kernel /boot/vmlinuz-$kernel root=$rootpart ro vga=normal'
>> /boot/grub/menu.lst
```

Make sure you change the line with “`root (hd0, 4)`” to fit your setup. With GRUB, you don't have to worry about applying changes to see the menu entry at boot. It's automatically there. With LILO, however, you have to actually apply changes each time you make them. You do this by running the `lilo` command as the superuser:

```
# lilo
Added Windows
Added Linux
Added 2.6.20-jcv1 *
```

The star (*) signifies the default kernel to boot. Make sure that your root partition is correctly specified in your boot loader configuration. My root partition is on `/dev/hda5`, but yours may be (and probably is) on a different partition. If you fail to specify the correct root partition, your system will not boot that kernel until the configuration is fixed. GRUB makes this a lot easier than LILO.

And this is the point when you start to cross your fingers and hope that your computer doesn't blow up... We get to reboot our computer and hope that our configuration file plays well with our computer. So, let's do that! See you in a few minutes (hopefully).

```
# shutdown -r now
```

So here I am, back on Linux on my freshly-rolled kernel. I hope you are as successful as I have been this time around. Keep in mind that you have to reinstall custom kernel modules if you installed others while you were on your other kernel. For example, I use `ndiswrapper` to access wireless Internet. I have to recompile and reinstall the `ndiswrapper` module and device drivers before I can use wireless. Likewise, I have VMWare Server on my laptop, which installed special modules. I have to run `vmware-config.pl` to reconfigure VMWare Server for my new kernel before I can run any virtual machines.

To summarize, here are the commands that I used in this tutorial. Remember that lines beginning with a dollar sign (\$) are executed as a non-privileged user, while lines beginning with the pound sign (#) are executed as the superuser (root).

```
$ cd /home/user/download
$ wget http://www.kernel.org/pub/linux/kernel/v2.6/linux-2.6.20.tar.bz2
$ su -
# cd /usr/src
# tar jxf /home/user/download/linux-2.6.20.tar.bz2
# rm /usr/src/linux
# ln -s /usr/src/linux-2.6.20 /usr/src/linux
# cd /usr/src/linux
# make clean
# vi Makefile (to change EXTRAVERSION to -jcv1)
# cp ../linux-2.6.19.1/.config .
# make silentoldconfig
# make menuconfig (just to ensure settings were good)
# time make
# make modules_install
# make install
# cd /boot
# fixkernelinstall 2.6.20-jcv1
# vi /etc/lilo.conf (to make sure things were good)
# lilo
# shutdown -r now
```

I hope that you are able to use this tutorial to successfully install or upgrade your kernel. Good luck! Any comments or suggestions are welcome!