

Kernel Rebuild Procedure

Kwan L. Lowe

8th February 2002

Why rehash the kernel rebuild? The documentation I've seen online does not have enough information in one place. This document consolidates and clarifies existing information. The online version of this guide is available at <http://www.digitalhermit.com/linux.kernel.html>.

Why rebuild the kernel? The main reason was once to optimize the kernel to your environment (hardware and usage patterns). With modern hardware there is rarely a need to recompile unless there is a particular feature of a new kernel that you must have. The performance gains are probably not noticeable unless specific benchmarks are being run. That said, recompiles are often necessary for situations with low memory (under 16M), low disk space (e.g., floppy or flash filesystems), or older hardware.

Preparation

2.1 Create a bootdisk

In the event that your kernel rebuild is catastrophic, the bootdisk will allow you to recover. Many of the CD distributions contain rescue and boot images, or the CDs themselves can function as a rescue disk. Insert a new floppy and format it with:

```
> fdformat /dev/fd0H1440
```

Your floppy drive may differ. If you receive an error about a missing device, try doing:

```
> ls /dev/fd*
```

This will list all possible floppy devices on your system. Try each of them in turn until you find the correct one. At the end of the formatting process the system will attempt to verify that the format was successful. It is not unusual to try four or five floppies before a "perfect" format is achieved. If you receive errors about the format, dispose of the floppy and use a new one.

Once formatting has completed successfully, run the `mkbootdisk` utility to create a rescue disk. This utility looks for a file in `/boot` called `vmlinuz-KERNEL_VERSION`, where `KERNEL_VERSION` is the output from the `uname -r` command. For example:

```
> uname -r 2.2.16-3
> mkbootdisk --device /dev/fd0H1440 2.2.16-3
```

Test your boot floppy before continuing.^([footnote]) The easiest way to get into trouble is to believe that you are flawless. Before literally dozens of kernel rebuilds, I had religiously checked my bootdisk. Guess what happened the first time that I didn't check it?)

2.2 Backup Modules Directory

Inside the `/lib/modules/` directory are the loadable modules for the installed kernel. You should see a directory with the same name as your kernel version. Do the following to backup this directory:

```
> cd /lib/modules
> ls 2.2.16-3/
> tar cvf modules.tar 2.2.16-3/
```

You will see a list of filenames scroll by as the directory is tarred. During the rebuild process the contents of the modules directories are modified. If they are deleted, they will be recreated.

Nervous about overwriting the modules directory? You can make a small change to the `/usr/src/linux/Makefile` file to change the kernel version. To do this, edit `/usr/src/linux/Makefile` and look for the following line:

```
VERSION = 2
PATCHLEVEL = 2
SUBLEVEL = 17 EXTRAVERSION =
Change the EXTRAVERSION line to reflect your changes:
EXTRAVERSION = KLL
```

In this case, I used my initials in the `EXTRAVERSION` field (vanity?). When the modules are created, the module directory will be:

```
2.2.17-KLL
```

Do not change the `VERSION`, `PATCHLEVEL` or `SUBLEVEL` fields as some applications may refer to these fields.

2.3 Verify correct versions of support software

The kernel requires many different packages in order to build correctly. For the most part, if you are running a recent distribution these packages are already installed. To make sure, read the `/usr/src/Linux/Documentation/Changes` file under the Current Minimum Requirements section. It will list the necessary packages and the command to verify them. E.g., from the `2.#.#` sources:

```
o Gnu C 2.91.66 # gcc --version
o Gnu make 3.77 # make --version
```

```

o binutils 2.9.1.0.25 # ld -v o util-linux 2.10o # fdformat --version
o modutils 2.4.2 # insmod -V
o e2fsprogs 1.19 # tune2fs
o reiserfsprogs 3.x.0b # reiserfsck 2>&1|grep reiserfsprogs
o pcmcia-cs 3.1.21 # cardmgr -V
o PPP 2.4.0 # pppd --version
o isdn4k-utils 3.1pre1 # isdnctrl 2>&1|grep version

```

Other packages that you may need include the ncurses-devel package if using "menuconfig" or tcl/tk if using "xconfig".

2.4 Install the Kernel Sources

Depending on your distribution, there are several ways of installing the kernel sources. For example, RedHat and Mandrake users can install the kernel-source and kernel-headers RPMs. The exact procedure varies depending on the distribution so please consult the distribution documentation for specifics. To use the menuconfig kernel configuration utility, you will also need the ncurses-devel package. Laptop users need the kernel-pcmcia package.

To install from the original kernel sources is somewhat lengthier and involves downloading the main source package and then applying the various patches. This method is necessary for the "bleeding-edge" kernels that have not been packaged. The newest sources are available from <http://kernel.org> or one of the many mirrors.

You do not need to download the full tarball if you have a previous version of the sources available. You may have noticed several patch files in the kernel sources download area. To update your older sources to the newest, grab all the patch files with versions higher than your source. For example, if you've already downloaded linux-2.4.11.tar.gz, download patch-2.4.12.gz to update your sources to the latest. The patches are not recursive, so to update from 2.4.11 to 2.4.14 means downloading three patch files.

Download the source and extract it to a directory where you have read/write access. Please note that the kernel documentation suggests that the sources not be installed into /usr/src as this can cause some problems elsewhere. Unfortunately, some program sources are hardcoded to look inside this directory for configuration files and may break if this is not the case. In addition, most distribution sources are installed by default into /usr/src/linux. If you're paranoid about installing or building anything as root, extract the sources into your home directory and build as a regular, non-root user. For the rest of this document, we'll assume that the sources are in /usr/src.

Kernel sources from the linux kernel mirrors are compressed in either gzip or bzip2 format. Depending on which version you downloaded, do the following:

```

GZIP: tar xfvz linux-VERSION.tar.gz
BZIP2: tar xfvj linux-VERSION.tar.bz2

```

After extracting the tarball, a linux directory will be created. You can now apply patches in the order of release using the patch utility. For example:

```
> cd /usr/src/linux
```

```
> gunzip patch01.gz
> patch -p1 < patch01
```

If your patch files are in gzip or bzip2 format, you can either extract them beforehand and run the above, or as the kernel README suggests:

```
> cd /usr/src gzip -cd patchXX.gz | patch -p0
```

OR

```
> cd /usr/src bunzip2 -dc patchXX.bz2 | patch -p0
```

(repeat xx for all versions bigger than the version of your current source tree, `_in_order_`) and you should be ok. You may want to remove the backup files (`xxx~` or `xxx.orig`), and make sure that there are no failed patches (`xxx#` or `xxx.rej`). If there are, either you or me has made a mistake.

Alternatively, the script `patch-kernel` can be used to automate this process. It determines the current kernel version and applies any patches found.

```
> cd /usr/src linux/scripts/patch-kernel
```

The default directory for the kernel source is `/usr/src/linux`, but can be specified as the first argument. Patches are applied from the current directory, but an alternative directory can be specified as the second argument.

In some cases you may need to get patches from third-parties. For example, the ReiserFS patches are not available in the stock 2.2.x kernel. In this case, run the patch against the version in the patchfile. For example:

```
> cd /usr/src/linux > patch < /tmp/linux-2.2.18-reiserfs-3.5.29-patch
```

To make the source directory as pristine as possible do:

```
> cd /usr/src/linux
```

```
> make mrproper
```

You should now have the sources correctly installed

Configuration

The configuration is the most intensive portion of the kernel rebuild. Don't worry, though. Unlike the kernels of yore, all newer versions have menu or graphical based front-ends. As mentioned previously, using these will require either `ncurses-devel` or an X-server (e.g. `XFree86` or `Metro-X`).

Before you configure your kernel, it's a good idea to save any existing configurations if they exist. Look in the `/usr/src/linux` directory for a `.config` file. Since it is a hidden file, use

```
ls -a
```

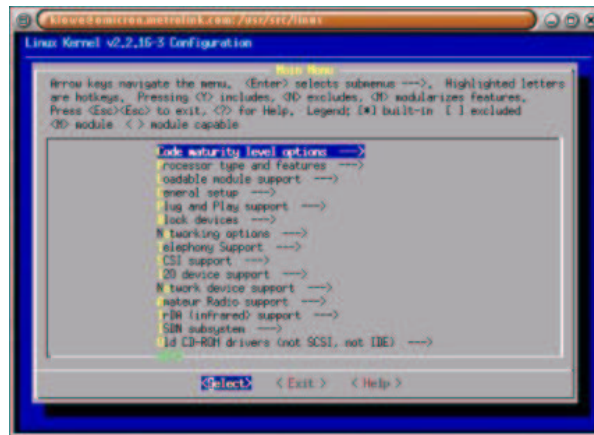
to list all files. If the `.config` exists, copy it to a new location. If you are using the sources from a distribution rather than the pristine sources from the kernel mirrors then the default configuration should be in `/usr/src/linux/arch/i386/defconfig`. If something goes wrong, copying this file over to the `./linux/.config` file should restore the distribution defaults.

Begin the configuration by typing:

```
> cd /usr/src/linux
```

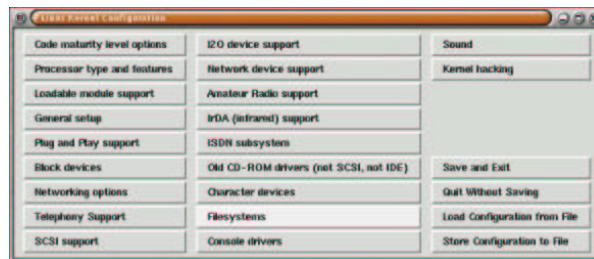
```
> make menuconfig
```

This will bring up a curses based screen similar to the following:



If you have X installed and functioning, you can run a tcl/tk configuration utility instead.

```
> cd /usr/src/linux
> make xconfig
```



Oddly enough, I have been able to use xconfig when menuconfig failed. Apparently there were problems with my curses libraries or terminal configuration that prevented the text menus from working. The curses library routines are a terminal-independent method of updating character screens with reasonable optimization.

If for some reason you cannot run the xconfig or menuconfig utilities, you will be forced to use the standard config. This method presents a series of questions to answer depending on choices for previous questions. Pressing [ENTER] will accept the default values. Unfortunately, there is no easy way to go back to a previous question without restarting. An example session may look like this:

```
Prompt for development and/or incomplete code/drivers (CONFIG_EXPERIMENTAL)
[Y/n/?] * Processor type and features * Processor family (386, 486/Cx486,
586/K5/5x86/6x86, Pentium/K6/TSC, PPro/6x86MX) [PPro/6x86MX]
```

If you choose make oldconfig the system will prompt and use the previous settings as the default values.

For the purpose of this document, we'll assume that make menuconfig is chosen. Many of the options can either be enabled or disabled, or built as a

module. Some options will enable or disable further related questions. For example, enabling experimental drivers will turn on the configuration questions for bleeding-edge development. If you choose to build modules, make sure that the modutils package is also installed to allow you to use the kernel module utilities.

Many options also have help text so specific information on all features is omitted here. To access the help is as simple as choosing the onscreen help button. This will usually provide information on the high-lighted option. Documentation is also stored in the `/usr/src/linux/Documentation` directory.

So how do you decide what modules to build into the kernel? Though building as modules saves some memory, it does take some time to load and unload the module. Depending on how your system is used, this extra overhead of loading and unloading modules may be unnoticeable.

Configuration Options

3.0.1 Code Maturity Level Options

This option enables configurations for the development or experimental features of the kernel. For example, in the 2.4.x series, ReiserFS is considered experimental, so enable this option to be able to use ReiserFS.

3.0.2 Processor Type and Features

The important option here is the Processor family. Read the help text to determine which option to choose. Linux distributors will generally build for the i386 architecture. Though this guarantees that the kernel will run on just about any x86 hardware, it is highly sub-optimal. In this section is also the option for math-emulation. Unless you are building on a 386 or 486sx, this is unneeded.

3.0.3 Loadable Module Support

3.0.4 General Setup

The main tweak here is Sysctl support. Sysctl is a means of configuring certain aspects of the kernel at runtime. In other words, you don't need to recompile the kernel to tweak it, thus obviating significant portions of this document! Refer to the `Documentation/sysctl` directory for more information on this option. If you are not using a laptop you may also disable the Power Management options.

3.0.5 Plug and Play Support

3.0.6 Block Devices

The main tweaks are the hard drive optimizations that can be enabled by default.

3.0.7 Networking Options

If you are configuring a router or server, pay close attention to these options.

3.0.8 SCSI Support

Some devices may require SCSI to be built into the kernel in order to boot.^[footnote]
Note: The `initrd` utility may help if you are having difficulty booting from a SCSI device. It gracefully (?) sidesteps the chicken and egg problem of not knowing how to read the root filesystem.) If you have an IDE-only system you can safely disable SCSI support. Performance tweaks include disabling profiling under the low-level drivers menu.

3.0.9 Network device support

Because Linux is highly network-oriented, many of these options are enabled by default by the distributors. Most of them are unneeded and are available as modules anyway. Disabling them probably won't do much for performance. One exception to this may be the actual driver under "Ethernet (10 or 100Mbit)". Building the driver into the kernel (rather than as a module) may improve performance on high usage networks.

3.0.10 Character devices

There's lots to disable here judging from the RedHat and Mandrake defaults. For a server system the watchdog timer may be useful to restart in the event of a lockup. You'll also find the configuration for I2C devices here. I2C is used by system monitoring software such as `lm_sensors`.

3.0.11 Filesystems

You should definitely build your root filesystem type statically into the kernel. Mandrake 7.1 was guilty of not building the ReiserFS into the kernel, but allowing Reiser to be selected as the root filesystem. The system would install without errors, but would panic on the first boot because the kernel would not know how to read the root filesystem to load the Reiser module. Under the Network Filesystems option you can also build in the NFS Server statically for improved performance.

3.0.12 Console Drivers

This one is not really a performance tweak, but building the framebuffer options can enhance console mode by enabling a high-resolution display for those cards that do not support `SVGATextMode` or look grainy with the `VGA=ASK` boot option. This might also be required if your video card is not directly supported (i.e., on LinuxPPC). Information on framebuffer devices can be found in the `Documentation/fb` directory.

Saving the Configuration

Once you have selected the modules you want, select exit from the main configuration menu. You will be asked if you would like to save your changes. Answer Yes and exit. I often backup my configuration using the Save Configuration to an Alternate File option. This makes it easy to retrieve a known working configuration later. You may also want to do this before you make any configuration change.

Creation

Next, run make dep and make clean. E.g.:

```
> make dep && make clean
```

Lots of messages will scroll by. Depending on the speed of your machine and on what options you chose, this may take several minutes to complete.

Next, start the actual kernel build by typing:

```
> make bzImage1
```

As the Kbuild documentation states:

Some computers won't work with 'make bzImage', either due to hardware problems or very old versions of lilo or loadlin. If your kernel image is small, you may use 'make zImage', 'make zdisk', or 'make zlilo' on these systems.

On an AMD K6-2 450, building the bzImage took approximately five minutes. On a Pentium 100, a similar configuration took almost 45 minutes. If you are not in a hurry you may want to start the build on a console while you continue to work. You don't need to compile the kernel on the machine on which it will be installed, though this is safest. As long as you select the correct features (use the saved configuration file) you can build it elsewhere.

When the build has finished the new kernel will be placed in the directory
`/usr/src/linux/arch/i386/boot`

Depending on which option you chose, it will be called vmlinuz, zImage or bzImage.

There is one more step needed for the build process, however. You have created the kernel, but now you need to create all the loadable modules. Do this by typing:

```
> make modules
```

Again, lots of compiler messages will appear as the modules are created. Other documentation usually instructs that the 'make modules_install' be run next. Don't do this until you have backed up the old version! For example:

```
> cd /usr/lib/modules  
> tar cvfz old_modules.tgz 2.2.16-3/
```

¹Note: the difference between 'zImage' files and 'bzImage' files is that 'bzImage' uses a different layout and a different loading algorithm, and thus has a larger capacity. Both files use gzip compression. The 'bz' in 'bzImage' stands for 'big zImage', not for 'bzip'!

The reason is that 'make modules' will overwrite files in your modules directory. As mentioned above, you can also edit the /usr/src/linux/Makefile and change the EXTRAVERSIONS line to create a new modules directory. If you are building the same kernel version as the currently installed version, then the modules should be identical. Run the modules installer with:

```
> make modules_install
```

4.0.1 Troubleshooting

If your build fails with a signal 11 error it is most likely because of hardware problems; often the culprit is failing memory. Unfortunately, the BIOS memory check is close to useless in detecting intermittent memory failures. Even dedicated memory checkers do not stress memory as much as gcc running a kernel build. One way to tell if hardware is at fault is to restart the 'make bzImage' process. If you can get a little further before failing again then it is a hardware error. There are several possible way to try to correct these.

Try changing your memory settings in the BIOS to more conservative levels. For example, change to SLOW or NORMAL instead of FAST. Verify that all the fans are working correctly.² Swap out the memory. One trick is to specify less memory than is actually installed by passing values to the kernel on boot. This prevents the kernel from accessing all the memory in the machine, and could help diagnose bad SIMMs or SDRAMs.

If instead the 'make' fails at the same point each time, then it is a configuration error. These usually result from not enabling a feature that is required by another. For example, IP Firewalling requires TCP/IP. If the prerequisite is not enabled, the build will fail. You may also get errors if you select the wrong processor or are using either a very old or development compiler.

Installation

Once your kernel is created, you can prepare it for use. From the /usr/src/linux directory, copy the System.map file to /boot.

```
> cp /usr/src/linux/System.map /boot/System.map-KERNEL_VERSION
> ln -s /boot/System.map-KERNEL_VERSION /boot/System.map
```

Next, change to /usr/src/linux/arch/i386/boot([footnote] On older systems or other architectures this will be elsewhere; E.g., /usr/src/redhat/linux/arc/i386/boot.) . This directory will contain the bzImage or zImage kernel. Copy this file to /boot with:

```
> cp bzImage /boot
```

²For a long while, I thought that the xmatrix screensaver was crashing my machine because of the numerous core dumps I would discover in my home directory. It turned out that xmatrix was cpu intensive. Unknown to me, the CPU fan on this machine had failed. Everything was fine until xmatrix started, causing the processor to overheat, eventually leading to a crash.

It's often helpful to rename the bzImage file to something more useful. For example, copy it to /boot/bzImage-2.4.4.

The next step is to configure lilo. Edit the /etc/lilo.conf file using your favorite editor. The highlighted text shows what was added or modified.

```
boot=/dev/hda
map=/boot/map
install=/boot/boot.b
vga=794 default=Testing
keytable=/boot/us.klt
lba32
prompt
timeout=50
message=/boot/message
menu-scheme=wb:bw:wb:bw
image=/boot/vmlinuz
    label=linux
    root=/dev/hda3
    append=" ide1=autotune ide0=autotune"
    read-only
image=/boot/bzImage
    label=Testing
    root=/dev/hda3
    read-only
```

The important sections are the `image=/boot/bzImage` and the `default=Testing` options. Notice that you can have several image sections in the lilo.conf, allowing multiple configurations.

Install the new kernel by running the lilo program.

```
> /sbin/lilo
```

Messages will appear showing the newly added kernel with an asterisk marking the default image. If you get errors, consult the lilo documentation for the correct syntax.

Reboot

The moment of truth is to shut down the machine and restart. Providing you listed it as the default, the new kernel should load. If you enabled an obvious feature such as the framebuffer console, you should be able to tell immediately if your new kernel works. Otherwise, look at the first line of the dmesg output to tell. E.g.:

```
> dmesg | head -1
Linux version 2.2.16-3 (root@omicon.metrolink.com) (gcc version egcs-2.91.66
19990314/Linux (egcs-1.1.2 release)) #13 Thu Dec 21 18:12:25 EST 2000
```

In this case, the kernel was built by the root user on the omicon.metrolink.com machine. As you boot you'll most likely see errors about undefined references. These occur because old modules exist in /lib/modules/KERNEL_VERSION but are not enabled in the kernel. Once you have tested the new kernel you can stop these messages by deleting the old modules directory and re-running 'make modules_install'. For example:

```
> cd /usr/lib/modules
> mv 2.2.16-3/ 2.2.16-3.old
> cd /usr/src/linux
> make modules_install
```

If for some reason the reboot fails, you have the tarball you created earlier for recovery.

6.0.1 Troubleshooting

Several things can cause the kernel to fail. Look at the error messages as the machine boots. A few things to check are: Is the root filesystem or the necessary SCSI drivers built into the kernel? Did you run LILO after installing the kernel? Did you build in the correct options for your motherboard? Did you choose the correct processor type for your CPU?

If your boot fails, restart and interrupt the default boot by pressing [TAB] at the LILO: prompt. This will allow you to choose an earlier kernel version.

As of this writing (September 6, 2001), there may be issues with the 2.4.x kernel and the amount of swap space. If you are upgrading from a 2.2.x series to 2.4.x, you will likely need to increase the amount of swap space available to the system. A good guideline is twice the amount of physical RAM. This does not mean that you need to repartition. You can try something like:

```
> dd if=/dev/zero of=/swap bs=1M count=128
> sync
> mkswap /swap
> swapon /swap
```

This will create a 128M swapfile on your root drive.

Acknowledgements

Links

www.kernelnewbies.org

References

- [1] Torvalds, Linus. "Linux Kernel 2.2.x documentation".
- [2] Chastain, Michael Elizabeth. "Overview of KBuild Commands". January 1999.
- [3] Gelinas, Jacques and Bjorn Ekwall. " modules.txt". August 1999.
- [4] Wolff, R.E. "Signal 11 while compiling the kernel".
<http://www.bitwizard.nl/sig11/>

- [5] Juhl, Jesper. "Compiling Your Own 2.0.x Kernel".
<http://jesper.staff.groundcontrol.dk/doc-linux-kernel-2.0.x/kernel.htm>