

Linux Kernel Compile Quick Reference Card

The Process

Downloading

The best way to download the kernel is to hit one of the mirror sites. You can visit <http://kernel.org/mirrors/> for a complete list of kernel mirror sites. Or you can connect to one of your closest mirrors by connecting to:

```
⇒ ftp://ftp.ca.kernel.org/pub/linux/kernel/v2.4, or
⇒ http://www.ca.kernel.org/pub/linux/kernel/v2.4.
```

Note that in both URL's the *ca* part can be substituted for another country code; again, see the mirror list for details.

Linux kernel releases take on the format of `<version>.<patchlevel>.<sublevel>-<extraversion>` where:

- ⇒ the version changes with major milestones,
- ⇒ the patchlevel changes with minor milestones,
- ⇒ an even patchlevel implies a stable line of kernels,
- ⇒ an odd patchlevel implies a development line of kernels,
- ⇒ the sublevel changes with bug-fixes, updates and optimizations,
- ⇒ while extraversion is used to identify pre-releases, test releases, and private development kernels.

A file listing of the above two URL's will reveal hundreds of files. There are two (more) important types of files:

```
⇒ linux-2.4.*.tar.*, and
⇒ patch-2.4.*.*.
```

Respectively these are whole kernel archives and patches between complete archives. Patch files are smaller to download and contain the delta between the version they bare in their name and the minor release just prior to it.

Validating

Along with any archive or patch file, one should download the corresponding cryptographic signature (*.sign*) file. A signature file can be used to authenticate that the kernel source archive, or patch file, is authentic. To do this you will need *gpg*.

Before you verify your first kernel you will need to get the "Linux Kernel Archives Verification Key". You do this by running:

```
▷ gpg --keyserver wwwkeys.pgp.net --recv-keys 0x517D0F0E
```

You can also get the PGP key from <http://www.kernel.org/signature.html>

Once you have a the key on your *gpg* *keyring*, and have downloaded the kernel file and its signature, you are ready to verify the authenticity of the download:

```
▷ gpg --verify linux-2.4.20.tar.gz.sign linux-2.4.20.tar.gz
```

gpg will notify you with success or failure of the test.

Unpacking

Unpacking the kernel is a very simple operation, you just have to use the *tar* utility. However some software may require the use of the kernel headers. Some software will blindly reference `/usr/src/linux`, while the kernel may be unpacked to `/usr/src/linux-2.4.20`. To fix this it is common to create a symlink for the most recent version of the kernel. These steps are shown below:

```
▷ cd /usr/src
▷ rm linux                    assuming linux is a symlink
▷ tar xzf /tmp/linux-2.4.20.tar.gz  creates a linux dir
▷ ln -s linux-2.4.20 linux     link new kernel as default kernel
```

If your archive has a *bz2* extension then you will need to uncompress them using *bunzip2* and then run *tar xf* instead. It is also common for *tar* on many distributions to have a *-j* option to uncompress *bz2* files just the way that *-z* decompresses *gz* files. If your *tar* utility has this feature you can run: `tar xjf linux.tar.bz2` instead.

Patching

If you have downloaded some patch file, say `/tmp/patch-2.4.21-pre1.bz2`, and wish to apply it to your kernel directory, `/usr/src/linux`, you should first make sure that the patch is compatible with your kernel sources:

```
▷ cd /usr/src/linux
▷ bunzip2 -c /tmp/patch-2.4.20-pre1.bz2 | patch -p1 --dry-run
```

If the output is full of patching file *foo* lines then it's successful. The *--dry-run* does not do make any changes but simply tests to see if the patch could be applied. Once you are ready to apply the patch for real run:

```
▷ cd /usr/src/linux
▷ bunzip2 -c /tmp/patch-2.4.20-pre1.bz2 | patch -p1
```

Note that if your patch file is *gzipped* and not *bzipped* you should run *gunzip* in place of *bunzip2*.

Configuration

The Linux kernel can be tailored to a specific computer it runs on; each driver and system component can be compiled into the kernel, compiled as a dynamically loadable component, or removed completely. Starting from scratch your kernel sources will have a default configuration that was picked by the maintainer. Chances are very slim that these defaults will match your system hardware; hence the configuration step. Most commonly you will run one of the following to configure your kernel:

```
▷ make menuconfig      a text-menu (curses) based config interface
▷ make xconfig         a graphical (X) based config interface
```

Most options in the kernel can be compiled into modules. See the Modules section to learn more about what should and should not be a modules. If the boot process requires a module you will need to create an *initrd* file (see below).

Configuration options are stored in a *.config* file. It is humanly editable, but quite overwhelming. If you have a *.config* file from a previous working kernel you can copy it into a new, and presumably unconfigured kernel, and run:

```
▷ make oldconfig      use a .config file as is
```

If you are bringing in a *.config* file into a previously configured or compiled kernel tree you should clean up the kernel source first. Frequently `clean` will not do the job; this is because there have been too many changes between the original configuration and the current. In these cases you will have to run `make mrproper` which will clean up the whole tree and even delete the *.config* file.

Compiling

The next step is to compile the kernel and produce the kernel and the module binaries.

```
▷ make bzImage        make a big, compressed kernel binary
▷ make modules        make all modules components
```

The order of the commands is arbitrary; you can build modules before you build the kernel. After compiling you will have a `/usr/src/linux/arch/i386/boot/bzImage` kernel binary and module files scattered through out the kernel tree.

If you are not building on an *i386* use the directory, under *arch* that reflects your architecture.

Installing

Finally, the kernel and modules are installed on the local system by running:

```
▷ make install        the kernel & System.map are installed in /boot
▷ make modules_install modules are installed in /lib/modules/<version>
```

If you compiled boot-critical functionality into modules then you will need to setup an *initrd* file; to do this run the *mkinitrd* program. The specifics of this differ from distribution to distribution. On RedHat for example you would run this:

```
▷ mkinitrd -f -v [options] /boot/initrd-<version>.img <version>
```

```
options: --fstab=file      determine file system modules needed using /etc/fstab
         --preload=module  module will be loaded before SCSI system modules defined in /etc/modules.conf
         --with=module     module will be loaded after the SCSI subsystem
```

On a recent Debian system, you could run the following after editing your `/etc/mkinitrd/mkinitrd.conf` - see the man page for details.

```
▷ mkinitrd -o outfile [moduledir]
```

An *initrd* is optional and usually not needed if you compile boot-critical components into the kernel and not

as modules.

Bootloaders

Bootloader is the glue between the BIOS and the kernel. Bootloaders allow the user to select what kernel, or other operating system, to boot. There are two commonly used boot loaders: *LILO* and *GRUB*. Check what your system uses; *LILO* config file lives in `/etc/lilo.conf`, while *GRUB* lives in `/boot/grub`.

menu.lst

```
title 2.4.20 \\  
root (hd2,0) \\  
kernel /boot/vmlinuz-2.4.20 ro root=/dev/sda1 vga=ext \\  
initrd /boot/initrd-2.4.20.img
```

lilo.conf

```
image=/boot/vmlinuz-2.4.20  
label=2.4.20  
initrd=/boot/initrd-2.2.20.img  
read-only  
root=/dev/sda1
```

Note that on RedHat the *GRUB* configuration file is named *grub.conf*.

After altering the *lilo.conf* file you will have to run the `/sbin/lilo` utility to “install” the configuration. *GRUB* requires no such step; it does need to be installed once – RTFM.

Files

Source

By convention the kernel source resides in `/usr/src/linux`. When multiple kernel sources reside on one machine it is customary to store them under `linux-<someversion>` directory and making `linux` a symlink to a *current* Linux kernel. An example:

```
$ ls linux* -ld  
lrwxrwxrwx 1 root root 17 Jan 10 19:21 linux -> linux-2.4.21-pre1/  
drwxr-xr-x 14 root root 4096 Mar 16 23:18 linux-2.4.18-pre9/  
drwxr-xr-x 14 root root 4096 Apr 10 19:21 linux-2.4.19-pre2/  
drwxr-xr-x 16 root root 4096 Jan 8 2002 linux-2.4.7-10/
```

The `linux` symlink is used to let other software packages which assume that they can locate the kernel source in `/usr/src/linux` to do just that.

Inside the `linux` directory we have the following:

Makefile	build rules for compiling the system
MAINTAINERS	list packages and their maintainers
REPORTING-BUGS	the right procedure for bug reporting
Documentation/	plethora of docs of varied vintage
arch/	architecture specific abstraction code stubs
drivers/	source tree of various drivers
fs/	file system implementation
include/	header files
init/	init code: main() of the kernel
ipc/	inter process communication implementation
kernel/	kernel core: scheduler, timers, signals, system calls, etc.
lib/	small primitives and utility functions
mm/	memory management: allocators, slabs, memory maps, shared memory, etc.
net/	networking code: routing, firewalling, etc
scripts/	scripts for build system, developer and user

The Makefile

The Makefile, and friends, have a variety of rules and targets that can be invoked. Here is a short summary:

clean	removes all binary files from the tree
mrproper	clean & removes all dependencies, architecture files, and documentation
distclean	mrproper & removes all patching relics
config	old style configuration; not worth using
xconfig	configuration through an X gui
menuconfig	configuration through a curses text-menu interface
oldconfig	uses existing <i>.config</i> to configure the kernel
bzImage	builds a 'big' gzip-compressed kernel
bzdisk	bzImage & copies it to a floppy
zImage	builds a gzip-compressed kernel
zdisk	zImage & copies it to a floppy
modules	build all modules
install	installs a bzImage in /boot
modules_install	installs all modules in /lib/modules
spec	creates an <i>rpm</i> spec file
rpm	rpm & builds the source rpm
htmldocs	build html docs in Documentation/DocBook/
pdfdocs	build pdf docs in Documentation/DocBook/
psdocs	build ps docs in Documentation/DocBook/
sgmldocs	build sgml docs in Documentation/DocBook/

Kernel

Once the kernel is compiled and installed you will have the following files on your system:

<code>/boot/vmlinuz-2.4.20.img</code>	the Linux kernel binary
<code>/boot/System.map-2.4.20.img</code>	map of kernel symbols and their memory offsets
<code>/boot/initrd-2.4.20.img</code>	<i>optional</i> , an initial ram disk file

Modules

Modules are installed into `/lib/modules/<version>` directory. This directory has a specific structure:

build	a sym links to the actual <code>/usr/src/linux-<version></code> directory
kernel	drivers and system component modules
pcmcia	pcmcia only drivers
misc	other (not from the kernel) modules
modules.dep	dependency list (used by <i>modprobe</i>)

Once the modules are installed there are a few utilities you can use to manage them:

depmod	build a dependency list; <code>/lib/modules/<version>/modules.dep</code>
insmod	install a module into a running kernel
modprobe	insmod a module and its dependents
rmmod	remove an installed module
lsmod	list installed modules

Bart Trojanowski <bart@jukie.net>
Dave O'Neill <dave@acm.org>