Chapter 4. The Kernel Sources (45)

Kernel Sources Structure

Linux is "Unix-like". Linux is not a version of Unix, but a new OS highly reminiscent of Unix. Linux aims to be Unix-compatible, but it has been developed from scratch. This allows the authors develop a new core framework which is highly portable. In the years since the first Motorola 68000 port for the Amiga, Linux has been ported to virtually every platform from the PalmPilot to the IBM S/390 mainframe.

Linux is also a large and complex piece of software. The kernel itself contains over 1.7 million lines of source code; by commercial software standards, such a project would take 5 five to 10 years and require up to 500 programmers. The methodology of open source has been proven through the kernel project: Richard Gooch's Kernel FAQ (http://www.tux.org/lkml/) observes that Linux is a little more than 8 eight years old, placing it right on track compared to those commercial standards.

Linux is written by thousands of programmers spread around the world, and under the peer-review of many thousands more. In the software industry, Brooks' Law claims the communications complexity with large teams will grow exponentially, eventually stifling all progress. The Linux project defeats this law by two clever hacks: Linux is organized into modules and subsystems where development in one area can have minimal impact on other areas, and the explosive growth and experimentation on new features is isolated from the mainstream users through the division of Linux into development and production sources. The modular design factors Linux into sub-teams and sub-teams of sub-teams, distributing the communications load, and avoiding the omni-topology that invokes Brooks' Law.

Figure 4-1. Package map of `/usr/src/linux`

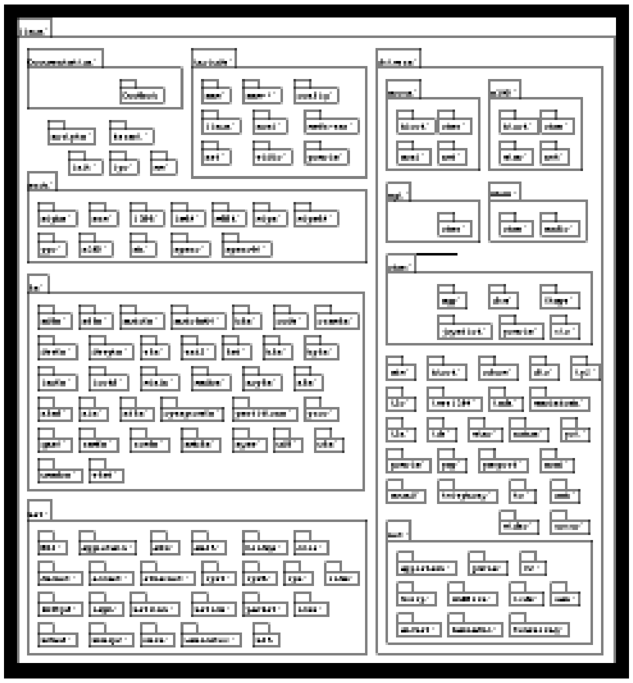


Figure 4-1 shows a package map of the kernel sources and the partitioning into subsystems. A quick inspection of `linux/MAINTAINERS` will show almost direct correspondence. The gross structure of Linux partitions the sources by technologies; since those technologies suit the fancies of the developers, someone with an interest in file systems may be deeply involved in the `/fs` package and only marginally involved in `/mm`.

The per-developer partitioning is carried deeper in per-service relationships within the `/net` and `/driver` packages. This makes sense when you consider each service is added and maintained by owners (or vendors) of that hardware who would want to keep their realm of interest localized. This "implementation-oriented" source structure may overlook similarities and opportunities for code re-use, but it is highly pragmatic within the wide-open open source methodology.

Kernel Modules

Configuring a kernel primarily involves selecting some devices and services and omitting others. Linux 2.0 introduced a middle ground allowing components to be dynamically loaded and unloaded from the kernel, as needed, at runtime.

Modules are also essential where the hardware must be initialized before a service is enabled. One example is with plug-and-play sound cards, where the boot process must initialize the interrupts before the module can be invoked.

The introduction of modules to the Linux kernel also has a political implication: Because module system calls are not considered “linking against the kernel”, modules are not bound by the GPL which that governs the rest of the kernel sources. Developers are free to create proprietary modules and to distribute these without releasing the source code. This policy has prompted several commercial vendors to provide modules where patent or other licensing issues would otherwise prevent Linux support.

Many runtime parameters, such as sound card ports, hard disk geometries, and IRQ assignments, can be set using “bootparams”, the command line options to the `LILO`. Also, many characteristics of the running kernel, even delicate issues such as virtual memory and filesystem behaviors, can be queried and set through the `/proc` filesystem (if `/proc` is enabled in your kernel).

Most kernel services accept bootparams; a complete guide to boot parameters can also be found with the **man bootparam** command.

In addition to the bootparams, some modules can be queried and manipulated through **echo**ing some value to a `/proc` file, for example, defense against Syn-Cookie attacks is enabled by appending the following line to `/etc/rc.d/rc.local`:

```
echo 1 >/proc/sys/net/ipv4/tcp_syncookies
```

Kernel Version Numbers

Linux kernel version numbers identify the base design and the revision, and also identifies whether you are running an experimental or a production release. The version in use by any Linux system can be queried with **uname -a**:

```
$ uname -a
Linux kato 2.3.21 #1 Tue Oct 12 16:08:21 EDT 1999 i486 unknown
```

This line identifies that kernel as version 2.3.21 and gives the architecture and the date when the kernel was compiled. The version number contains three parts:

- The major number (2)
- The minor number (3)
- The current revision (21)

There are enough version-1 Linux machines to be impressive, but they are rare. For practical purposes, the major number is 2; no one is quite sure what a change would be big enough to jump to Linux-3, but we might assume it would cease to promise Linux-2 compatibility.

The last number is the incremental patch number and changes with every update. The first important portion of the kernel version number is the middle digit, the “minor” number. *Odd* minor numbers denote “experimental” kernels, the testbed for developers to explore, add new code and bounce ideas off each other. *Even* minors, on the other hand, carry a stamp of respectability as “production editions”.

When a development kernel is ready, it follows the usual process of a feature freeze, then a code freeze, and is then promoted to the next even number. Thus, 2.3.99 became 2.4.0 and was simultaneously branched to become the new development 2.5 source. The road goes ever on.

Newsflash: Development Kernels can break

During the rapid development of the odd-minor kernels, you cannot simply jump in and expect to join the other kernel surfers. Although developers try to only release stable code, very often a core change can create havoc for other users. These troubles may be traced to updated libraries, modules, or compiler tools, but sometimes it is just a broken system and the development kernel can strand whole communities of users. Always check the Linux Kernel web site (http://www.tux.org) or watch `linux-kernel` mailing list. If your hardware is not flagged in the trouble reports, pick a development kernel revision which has been stable for at least a few days, and dive in.

Obtaining the Kernel Sources

When fetching kernels sources, it is good nettiquette to seek a mirror site as close to you as possible. While famous sites such as Red Hat (ftp://ftp.redhat.com) and Kernel.Org (ftp://ftp.kernel.org/pub/linux) will have the files you need, you should check kernel.org (http://www.kernel.org) for an appropriate mirror site.

Distribution Kernels

Before you replace your kernel sources, be aware that most distributions apply their own patches to the standard Linux sources; if you recompile from your distro sources, there may be features missing or added from what is covered in this chapter, and if you obtain “canonical” sources from the kernel FTP site (ftp://ftp.kernel.org), the resulting kernel may break special features in your distribution.

FTP mirrors partition kernel packages by major and minor version numbers such as `/pub/linux/kernel/v2.4`. Each directory contains whole-enchilada source tarballs and compressed patch files — if you are jumping up from 2.2 or 2.3, or if you want to start fresh, you need the file named `linux-VERSION.tar.gz` (or `.bz2`). Although a lot to download over a modem connection, once you have a standard source tree, you can update incrementally by downloading much smaller patch files.

Symbolic Links to the Kernel Sources: Before you unpack a kernel tarball, create a version-tagged directory and symlink (**ln -s**) this to the generic `/usr/src/linux` location. For example, if you obtained `linux-2.4.7.tar.gz`, create `/usr/src/linux-2.4.7` and a symlink `/usr/src/linux` before you unpack your kernel sources:

```
cd /usr/src
mkdir linux-2.4.7
ln -s linux-2.4.7 linux
tar xzf linux-2.4.7.tar.gz
```

If `/usr/include/linux` is also symlinked to `/usr/src/linux/include`, your include files will always belong to your current kernel. You can now keep several versions of the kernel, each in it's own `/usr/src/linux-X.Y.Z` directory; when you patch your sources, the patch will apply to the appropriate kernel sources through the `linux/` symlink, and you can experiment with multiple versions, quickly switching between them by simply replacing the symlink.

Coping with New Kernels

- Building and running the kernel depends on the software versions listed in `linux/Documentation/Changes`: Before you compile or run any new kernel, you can save a lot of heart-ache if you check your system against these version numbers!
- Drivers bundled with Linux may not be the very latest. If you have problems with a particular device, or with very new hardware, search for an update before building your kernel.
- Kernel changes may require changes to your boot scripts, to `/etc/lilo.conf`, or `/etc/conf.modules`. The `linux/Documentation` collection contains many short `README` files on many different parts of the kernel, and each driver subdirectory may also contain additional information on installing or configuring difficult devices. Most kernel modules accept parameters either through the boot params, through the append line of `/etc/lilo.conf`, or, for dynamically loaded modules, on the **/sbin/insmod** command line or in `/etc/conf.modules`.
- If you give options both in `lilo.conf` and at the `LILO:` prompt, the boot prompt options are *appended* to the end of the `append` options. This allows you to override installed options at the command line to pre-empt unwanted settings; this is also why many modules also include options to restore their default behavior.

Patching the Source Tree

Updates are distributed as **patch** files named for the version which results from the patch. For example, an upgrade from 2.4.4 to 2.4.5 will be called `patch-2.4.5.gz`. Patch files are simply context diffs, the output of the CVS or Unix **diff -c** command (see **man diff**) which list the differences from the prior version. The patch is applied by piping the file through the GNU **patch** utility; **patch** will be given lines of context around the change and told to delete, add, or replace lines in the source files.

Patching Incremental Versions

Patch files can only upgrade by one revision number. Upgrading from 2.4.1 to 2.4.6 will require patches for all intermediate releases. An exception to this rule is the `-ac` series pre-release patches which are assembled by Alan Cox. These patch files are always accumulative based on the most recent official release, for example `patch-2.4.1-ac12` might be Alan's twelfth patch on the 2.4.1 release, but it must be applied to a clean 2.4.1 kernel; the patch will fail to upgrade from `patch-2.4.1-ac11`.

Unless you have good reason to suspect otherwise, you should compile and test each incremental revision before applying the next patch. Once a patch is applied, there is no way to undo the changes.

Patch files are also created by comparing the sources of totally clean Linux distributions. Before you run the patch, it is important to back up your `linux/.config` file and then to clean your kernel source tree to the most pristine state using **make mrproper**:

```
cd /usr/src/linux
cp .config /usr/src/config-old
make mrproper
```

This removes all residual generated files that could interfere with the patch or the kernel build. Saving your old `.config` (which would be removed by **mrproper**) allows you to bootstrap your next kernel with **make oldconfig**.

Two things you should know about patch files: They are much smaller than the source tarball (typically a few hundred kilobytes), and they do not always work.

Verifying a patch update: After patching your sources, search the kernel tree for reject files (**find . -name "*.rej"**) containing the failed diffs; these are most often the result of some innocuous difference in whitespace or tab stops between the sources owned by the creator of the diff and your own sources. If you find any `*.rej` files, you must *manually* correct the associated source files before you compile.

The `linux/scripts/patch-kernel` is a perl program which attempts to automate this process. **patch-kernel** will deduce the current source version number and compare this to the patch files found in the current directory. If higher-version patch files are found, the script will step through these sequentially upgrading the sources. Frankly, I have never had much luck with this script, but your mileage may vary.

Once you have patched the sources, you can bootstrap your new kernel configuration by copying your backup `.config` file to `linux/.config` and using **make oldconfig** to migrate the options present in your previous installation, preserving all those fiddly settings that are so often forgotten (such as the IRQ of your sound card!). You will need to monitor this process: **oldconfig** will stop and prompt for any new options added by the new sources. Once this command has run, use **make xconfig** to recheck the settings, and then compile with any of the **make** build targets.

Upgrades and Modules

One of two module things can go wrong with a module when upgrading to a newer kernel[1]:

The module utilities are incompatible.
The module dependencies may conflict.

The first situation is most likely. The ix386 modutils have been updated to use PIII instructions and new editions may not have been included in your distribution files. If your module utilities meet the requirements spelled out in `linux/Documentation/Changes`, you will have no problem, but if you do happen to boot a kernel with old module utilities, you can hang your system.

The second problem can hit anyone. This is one of those things which, once you know what has happened, you realize the solution is just common sense. For example, suppose you compile Windows VFAT file system support as a module, but then your boot scripts try to use some file from your windows partition prior to the module loading. Or, and more likely, you configure your network support as a module, forgetting that the **httpd** will hang the boot scripts while trying to resolve the hostname.

There is only a small overhead in loading and unloading, and in the coding of a driver as a module. Unless memory or performance is critical, the code is needed continuously, or the module is required early in the boot process, it can and probably should be compiled as a module.

Configuring Linux Kernel

Once you know what your new kernel must accomplish and have the sources installed, you are ready to begin creating a customized Linux operating system. Before setting the compile in motion, there are a few last minute details to ensure a successful run, or at least a graceful recovery.

Preparing lilo.conf for kernel updates

Before you begin, double check your system against the requirements in `linux/Documentation/Changes`. In particular, ensure you have the correct version of the binutils and the gcc compiler.

You should also create an entry in your `/etc/lilo.conf` to keep your current kernel installed as a backup. In addition to adding a few seconds delay on the boot prompt with the `delay` parameter (so you can interrupt to add parameters or select an alternate image) add a section for a known stable kernel (such as the original distribution kernel image) and also for the generated backup left behind by the build commands.

```
image=/boot/vmlinuz.orig
label=stable
root=/dev/hda3
append=""
read-only
image=/boot/vmlinuz.bak
label=backup
root=/dev/hda3
append=""
read-only
```

If you have trouble booting from the new kernel, you can enter **`stable`** at the `LILO:` prompt and boot your original kernel.

The **make bzlilo** compile command will automatically backup the previous kernel from `$(INSTALL_PATH)/vmlinuz` to `$(INSTALL_PATH)/vmlinuz.old` and then run the **lilo** command to install the new kernels. The above `/etc/lilo.conf` sections give one more line of defence against a kernel which cannot boot. A fourth section, labeled `backup` also allows me to make periodic backups of particularly stable development kernels in case repeated compiles leave both `vmlinuz` and `vmlinuz.old` unstable and utility upgrades have cut me off from the original `vmlinuz.orig`. It happens.

Is `lilo.conf` prepared to find the new kernel? Do you have a backup kernel and a boot disk? Do you have enough disk space? These may seem trivial questions, but they are important. An error in any of these may leave your system in an inoperable state and entirely devour your weekend. Even if all you are doing is to set a few `/proc` values or add a network interface on the boot prompt, it is good defensive driving to consider the recovery plan.

Allow a LILO Delay

While experimenting with kernel options, remember to set the lilo `delay` parameter to give some grace time where you can select an alternate kernel. If your system has a delay of zero seconds, you will not have the opportunity to pre-empt loading the default kernel, and if that kernel is faulty, your only option is to boot from a floppy disk. For more dangerous experiments, `delay` set to -1 will wait indefinitely for a boot parameter.

Configuring with make

The Linux Makefile provides four methods of setting your configuration options:

make config

A command-line terminal program

make menuconfig

An ncurses-based console program

make xconfig

A tk/tcl-based X11 GUI program

make oldconfig

A semi-automatic update program

Why have a command-line interface at all? Suppose you had an autonomous robot submarine, or a space probe. Imagine you are in the midst of maneuvers and needed a fast kernel reconfig. Simple dumb terminal interfaces can go places other interfaces cannot dream of going! The X11/tk interface may be more elegant and esthetically appealing, but it does require that Tk and X11 are both installed and working.

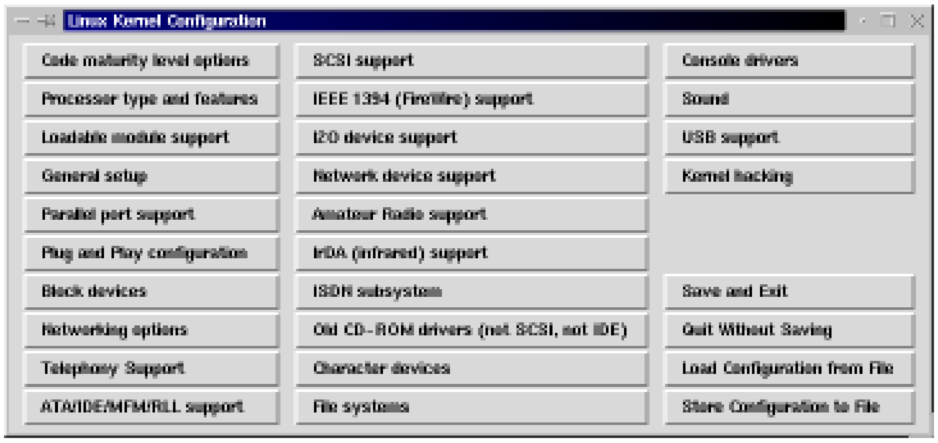
X11 is sometimes impractical, for example, while doing remote administration over a slow telnet connection or in a Linux machine for blind users. For these situations, we have the ncurses based `menuconfig`. Both the X11 and the ncurses configuration tools offer the same options in the same order, and they have roughly the same capability to navigate backwards and forwards through the configuration options[2].

To start the configuration, simply go to the `/usr/src/linux` directory and enter one of the configuration commands. If you choose either of the ncurses or X11 methods, you will see a brief flurry of compiler activity while the user-interface programs compile, and then you will be greeted by an overview screen with all the categories of kernel options.

Configuration Options

Figure 4-2 shows the initial screen as seen in the `xconfig` display for the 2.4 kernel. 2.2.x kernels are very similar, but with fewer options and minor cosmetic differences. In all of the configuration methods, most kernel options can be set to be included, included as a module, or left out of the compile. On the `xconfig` and `menuconfig` screens, there are options to include or exclude complete sections of configuration; disabling these will grey out any dependent options (the dumb-terminal config option will silently skip these sections).

Figure 4-2. screenshot of make xconfig screen



Whether you use the dumb-terminal, ncurses, or X11 method, and whether or not you are a beginner or advanced user, the most important feature on all the kernel configuration screens, is the HELP option. Almost all kernel features are documented right in the configuration screen; whether you are looking at options for installing sound or network support or seeking expert options for filesystems and firewalls, most options carry very reassuring advice: “It is safe to say Y” (or N).

> **Preparing for a Kernel Configuration:** To answer many of configuration questions, you must know the insides of your computer. Kernel configuration will ask about network cards, sound cards, PCI chipsets, IDE and SCSI controllers, and a host of other highly personal questions. When in doubt, use the defaults or the recommended option in the associated HELP page. If you’re feeling zealous, keep your computer manuals nearby or run through your first configuration with the panels taken off your computer and a flashlight in hand.

It is not absolutely essential that you match your computer chipset perfectly on your first kernel configuration. The defaults are normally work very well.

Kernel Hacking

As of Linux 2.2, `Kernel Hacking` contains only one option: To toggle SysRQ support. `SysRQ` adds several very useful commands for recovering from a hung system through binding recovery and diagnostic operations to Ctrl Alt SysRq key sequences. For example, if and the console or X-server becomes locked because some renegade process is blocking all I/O, you might try to telnet to the machine in hopes of opening a super-user shell to kill that process or reboot the machine, but if that fails, `SysRQ` can be used to **sync** and **umount** the filesystems, and to force a reboot. See Table 4-1.

Security and SysRQ

Kiosks, workstations and production machines should disable `SysRQ`; you may want to also disable the Ctrl Alt Backspace command to exit X-Windows. This prevents novice (or knowledgeable) users from bringing down the machine without authorization. In `/etc/inittab`, you can also customize the Ctrl Alt Del reboot interrupt to give a longer grace period or to disable the command entirely.

Table 4-1. Kernel SysRQ Commands

r	Turns off keyboard Raw mode and sets it to XLATE;. This is useful when the console or the X-Server is hung.
k	Kills all programs on the current virtual console. Use this to shut down a locked X-server.
b	Immediately reBoots the system without synching or unmounting filesystems. This command may corrupt your file-system if you have not already synch’ed and unmounted your disks.
o	Shuts Off system power via APM (if configured and supported).
s	Sync all mounted filesystems to minimize the filesystem corruption whichthat may occur from an ungraceful shutdown.
u	Unmount and remount all filesystems as read-only, much like the shutdown command. This allows your system to read the binaries required for an orderly shutdown.
p	Dumps the current registers and flags to your console (i.e. generates a kernel Panic).

t	Dumps a list of current Tasks and their information to your console, to givinge you the diagnostic details for to isolateing the cause of the hang.
m	Dumps current Memory info to your console.
0-9	Sets the console log level that filters kernel messages . For example, a level of zero 0 would filter out everything except panics and oops messages.
e/i	Sends tErm or kIll signals to all processes except init, effectively throwing you into single-user mode.
l	Sends `SIGKILL to all processes, including init, which effectively halts your system.`

Alternate Configurations

The `Load/Save` options are a convenience for those who need to maintain several alternate configurations, for example on a machine used to compile kernels for other machines, or where you need alternate kernels for different purposes. As you would expect, this option pops up a dialog, asking for the filename and then saves `.config` to the named location.

Saving Your Configuration

Once the kernel is configured, `save and exit` creates `.config`, and, if the kernel has been configured for sound, generates `linux/include/linux/autoconf.h`. The kernel is now primed and ready for building.

CML2: The Next-generation Configuration Tool

While the configuration scripts which drive the makefile do the job with a fair degree of assurance, this system was never intended to manage a project with close to two million lines of code and hundreds of interdependent options. To ease this situation, Eric Raymond has devoted his airtravel and hotel time to creating a next-generation solution, the “Configuration Management Language”, or CML2

Although CML2 was not ready for the 2.4 kernel, it will be included in 2.5 as an alternative configuration method and will either become the standard by 2.6 or will be superceded by some other CML. As much pain as this causes in maintaining parallel versions, the pay-off for Linux 2.6 and on will be considerable.

The current CML2 will handle all kernel configuration options, and will report and prevent incompatible or inconsistent configurations. The design of this new scripting language may also allow for someday configuring the kernel in reverse, by specifying the hardware on a form and having the CML2 interpreter deduce the correct kernel options, a large step towards someday having kernel configuration proceed automatically from the hardware detection stage.

Example 4-1. Using CML2

In most cases, migrating configuration specs to CML2 only involves patching the main rules file `kernel-rules.cml`, adding the symbol declarations to `kernel-symbols.cml` and ensuring the correct entry in `kernel-menus.cml`. Once your rules have been entered into these files, configuration specs can be tested and debugged using the **cmlcompile.py** utility to display the menu tree or to run the compiler in interactive mode.

`kernel-rules.cml` specifies all options with their sub-options and required components. For example, the menu rule for `Network File Systems` states

```
unless INET suppress CODA_FS NFS_FS SMB_FS NFSD
unless (IPX!=n or INET!=n) suppress NCP_FS
...
menu nfs # Network file systems
        CODA_FS? NFS_FS? {NFS_V3 ROOT_NFS} NFSD? {NFSD_V3}
        SMB_FS?
        NCP_FS? {ncpfs}
...
menu ncpfs # NCP filesystem configuration
        NCPFS_PACKET_SIGNING NCPFS_IOCTL_LOCKING
        NCPFS_STRONG NCPFS_NFS_NS NCPFS_OS2_NS NCPFS_SMALLDOS
        NCPFS_MOUNT_SUBDIR NCPFS_NDS_DOMAINS NCPFS_NLS NCPFS_EXTRAS
...
unless NCP_FS!=n suppress ncpfs
...
derive NLS from JOLIET==y or FAT_FS!=n or NTFS_FS!=n or NCPFS_NLS==y
```

This provides all five different network file systems when `INET` is enabled, but only offers the `NCP_FS` if `IPX` is set but`INET` is not. Where `NCP_FS` is enabled, the configuration will include the related `NCPFS_*` options. Also, `NCP_FS` will imply including the `NLS` code page support.

`kernel-symbols.cml` defines the pretty-print strings for the kernel symbols and is factored out of the rules file to allow for translations. This file holds no surprises:

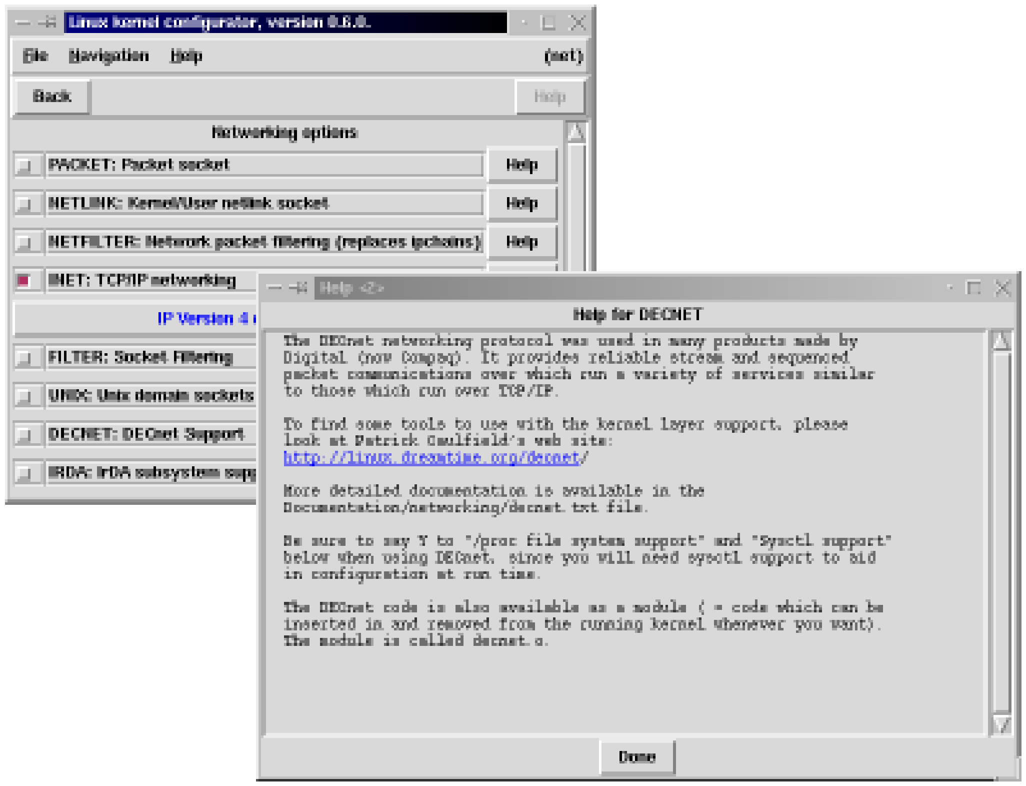
```
NCP_FS          'NCP file system support (to mount NetWare volumes)'
#
# NCP Filesystem configuration
#
NCPFS_PACKET_SIGNING    'Packet signatures'
NCPFS_IOCTL_LOCKING     'Proprietary file locking'
NCPFS_STRONG            'Clear remove/delete inhibit when needed'
NCPFS_NFS_NS            'Use NFS namespace if available'
```

```
NCPFS_OS2_NS            'Use LONG (OS/2) namespace if available'
NCPFS_SMALLDOS          'Lowercase DOS filenames'
NCPFS_MOUNT_SUBDIR      'Allow mounting of volume subdirectories'
NCPFS_NDS_DOMAINS       'NDS authentication support'
NCPFS_NLS               'Use Native Language Support'
NCPFS_EXTRAS            'Enable symbolic links and execute flags'
```

Similarly, `kernel-menus.cml` simply defines the top-level menus for the kernel configuration process.

Kernel configurations are generated in two stages, first to compile the menu source file with the **cmlcompile.py** program, and then to run **cmlconfigure.py** to set kernel options. Like the `Makefile` rules it replaces, CML2 offers X, curses and line oriented interfaces selected by the default mode or with the `-c` and `-t` options:

```
cmlcompile.py kernel-rules.cml
cmlconfigure.py
```


Linux kernel configurator, version 0.6.0.
File Navigation Help (net)
Back Help
Networking options
PACKET: Packet socket Help
NETLINK: Kernel/User netlink socket Help
NETFILTER: Network packet filtering (replaces ipchains) Help
INET: TCP/IP networking
IP Version 4
FILTER: Socket Filtering
UNIX: Unix domain sockets
DECNET: DECnet Support
IRDA: IrDA subsystem supp
Help <2>
Help for DECNET
Done

```
The DECnet networking protocol was used in many products made by
Digital (now Compaq). It provides reliable stream and sequenced
packet communications over which run a variety of services similar
to those which run over TCP/IP.

To find some tools to use with the kernel layer support, please
look at Patrick Caulfield's web site:
http://linux.dreamtime.org/decnet/

More detailed documentation is available in the
Documentation/networking/decnet.txt file.

Be sure to say Y to "/proc file system support" and "Sysctl support"
below when using DECnet, since you will need sysctl support to aid
in configuration at run time.

The DECnet code is also available as a module ( = code which can be
inserted in and removed from the running kernel whenever you want).
The module is called decnet.o.
```

Screen shot of CML2 v.0.6.0

CML2 is coded in the Python 1.5.2 to ensure portability and to allow direct compilation to a native binary. A complete guide to CML2 and the current source files can be downloaded from Eric Raymond's KBuild Webpage (http://www.tuxedo.org/~esr/kbuild/).

Building and Installing the Kernel

You can now apply your updated configuration to build the new kernel, create the modules, and install the works. It's also now time to give your wrists a break and return to the command line to put it all together.

Before building the new kernel, you must regenerate all dependency files to account for any changes in include or module options. Until CML2 becomes the standard, this step is needed whenever the kernel configuration is changed. The **make dep clean** command will rebuild these dependency files and remove any stray generated files.

Building the Kernel

As with everything else about Linux, building the kernel offers many choices. All the build options are outlined in `linux/Documentation/kbuild/commands.txt`; the most common cases are building just the kernel file, only the modules, building both, and doing an install of the kernel and or the modules after the build.

For example, you may need to build a kernel file for some other computer (such as a laptop) or to be installed by hand under some very logical new name. Building this image could be done with

```
make bzImage && \
    cp /usr/src/linux/arch/i386/boot/bzImage \
       /boot/vmlinuz-scsi
```

The first command compiles and compresses the kernel (for a "large kernel" image; see) leaving the result in `arch/$(ARCH)/boot`; the image is then copied manually to the `/boot` directory. This new image will not be installed until it is registered with a boot loader such as **lilo** (see the section called *Manually Installing a New Kernel*. To automatically install a kernel image, use the command

```
make bzlilo
```

This will create the same compressed image and copy the file to `$INSTALL_PATH/vmlinuz`, then run the lilo command; **make bzlilo** assumes your `/etc/lilo.conf` expects this filename in this location.

Big Kernels on Intel Machines

Each of the kernel build commands has a "big-kernel" counterpart that is needed if the kernel grows to be over 1MB in size when uncompressed. Installing these large kernel images with older versions of **lilo** will overwrite part of the boot loader and the system will not boot. Since most configurations result in a big kernel and building small kernels with the big-kernel commands seems benign, unless you are building a very simple kernel for an old machine or an embedded device, router, etc., use `bzlilo` and `bzImage` **make** targets just to be sure.

All `z*` targets are specific to the ix386 platform, and these will be removed during the 2.5 development and replaced by the `bz*` equivalents. For this reason, this chapter will use the normal `z*` commands.

The most common and convenient command for creating a new kernel is

```
make dep clean zlilo modules modules_install
```

This one command line will:

- Perform the dependency file generation.
- Clean the sources.
- Create a compressed kernel image.
- Move `$(INSTALL_PATH)/vmlinuz to $(INSTALL_PATH)/vmlinuz.old, copy the new zImage kernel fil`
- Build all modules and install these under `$(INSTALL_MOD_PATH)/lib/modules/2.2.5.`

Putting all these commands in on one line will ensure that if any stage of this build fails, subsequent stages will not be started. The whole process can also be scheduled to run in an xterm window or alt-console, can be run during off-peak hours as an **at** job, or can be used as an excuse to play some serious **Nethack** or **XPilot**.

Manually Installing a New Kernel

The new kernel file is always found in `/usr/src/linux/arch/$(ARCH)/boot/zImage` and must be installed using the **lilo** boot loader or some other Linux loader before it can be used.

For example, to emulate the Red Hat `/boot` path scheme, you would need to copy the new `zImage` to `/boot/vmlinuz` (save the old one first!) and modify `/etc/lilo.conf` to include the backup version. Alternatively, to accommodate stubborn plug-and-play devices, you may need to copy this new kernel to your Windows 95 partition for use by the Linux **loadlin.exe** boot loader.

One frequent requirement is to create boot floppies. A boot floppy is nothing more than a kernel copied directly to the a floppy disk and set to mount the root file system from the hard drive.

While it is far more mnemonic to create a boot floppy using the command

```
make zdisk
```

this is equivalent to using **dd** to copy the file directly to the raw sectors of the floppy disk device:

```
dd if=arch/i386/zImage of=/dev/fd0
```

> **Compiling for a Remote Machine:** When you're creating a kernel for some other machine (eg such as a laptop), you can create put the compressed kernel file and all modules into an alternate directory tree by giving alternate values for the `INSTALL_*` path variables on the **make** command line, for example:
>
> ```
> INSTALL_PATH=/psitta \
>     INSTALL_MOD_PATH=/psitta ROOT_DEV=/dev/hda1 \
>         make bzlilo modules_install
> ```
>
> This will move the generated kernel, map, and module files to `/psitta/vmlinuz`, `/psitta/System.map` and `/psitta/lib/modules`, where you can conveniently tar the whole directory for shipment to the remote machine with the following:
>
> ```
> cd /psitta && \
>   tar cf - vmlinuz System.map lib | \
>      rsh psitta tar xCf / -
> ```
>
> There is one side effect: The command will also run **lilo**, but since the `/etc/lilo.conf` file does not reference these new `/psitta` files, there is no adverse effect.
>
> The only potential hazard of this trick is a possible change to files in `/usr/include/linux`, which may affect programs subsequently compiled on the build host. Some care must be taken to ensure that the alternate kernel build does not leave unwanted changes in this directory on the build machine. Also, if the remote machine will be used to build software, after compiling the new kernel, you should copy the `/usr/include/linux` directory should also be copied over installed onto the remote machine after compiling the new kernel.

Troubleshooting the New Kernel

`/proc` is your friend. The pseudo-files in the `/proc` directory hold a wealth of diagnostic information and a simple means to set runtime parameters.

System Information Files

The most frequently useful `/proc` diagnostic files are as follows:

- `/proc/cpuinfo` lists the processor type, number of ports, and other essential information about the computer hardware:

  ```
  $cat /proc/cpuinfo

  processor : 0
  vendor_id : AuthenticAMD
  cpu family : 5
  model : 8
  model name : AMD-K6(tm) 3D processor
  stepping : 12
  cpu MHz : 350.804507
  fdiv_bug : no
  hlt_bug : no
  sep_bug : no
  f00f_bug : no
  fpu : yes
  fpu_exception : yes
  cpuid level : 1
  wp : yes
  flags : fpu vme de pse tsc msr mce cx8 sep pge mmx 3dnow
  bogomips : 699.60
  ```

- `/proc/interrupts` maps IRQ lines to devices:

  ```
  $ cat /proc/interrupts

  CPU0
  0: 30200579 XT-PIC timer
  1: 251230 XT-PIC keyboard
  2: 0 XT-PIC cascade
  4: 996021 XT-PIC serial
  5: 1 XT-PIC soundblaster
  7: 2 XT-PIC parport1
  8: 1 XT-PIC rtc
  11: 3984 XT-PIC MSS audio codec
  12: 973494 XT-PIC eth0
  13: 1 XT-PIC fpu
  14: 4253923 XT-PIC ide0
  15: 4713361 XT-PIC ide1
  NMI: 0
  ```

- `/proc/sound` reports the current sound system configuration and the installed services:

  ```
  $ cat /proc/sound
  ```

```
OSS/Free:3.8s2++-971130

Load type: Driver compiled into kernel
Kernel: Linux maya.dyndns.org 2.2.5 #2 Thu Apr 15 18:34:07 EDT 1999 i586
Config options: 0

Installed drivers:

Type 10: MS Sound System
Type 27: Compaq Deskpro XL
Type 1: OPL-2/OPL-3 FM
Type 26: MPU-401 (UART)
Type 2: Sound Blaster
Type 29: Sound Blaster PnP
Type 7: SB MPU-401
Type 36: SoftOSS Virtual Wave Table

Card config:
SoftOSS Virtual Wave Table
Compaq Deskpro XL at 0x530 irq 11 drq 0,0
Sound Blaster at 0x220 irq 5 drq 1,5
(SB MPU-401 at 0x330 irq 5 drq 0)
OPL-2/OPL-3 FM at 0x388 drq 0

Audio devices:

0: MSS audio codec (SoundPro CMI 8330)
1: Sound Blaster 16 (4.13) (DUPLEX)

Synth devices:

0: SoftOSS
1: Yamaha OPL3

Midi devices:

Timers:

0: System clock
1: SoftOSS

Mixers:

0: MSS audio codec (SoundPro CMI 8330)
1: Sound Blaster
```

- `/proc/parport` contains directories for each parallel port and reports on the devices attached to each port:

  ```
  $ cat /proc/parport/0/hardware

  base: 0x378
  irq: none
  dma: none
  modes: SPP,ECP,ECPEPP,ECPPS2
  ```

Setting Kernel Parameters and Options

Kernel and other low-level runtime parameters can be set through the `/proc/sys` pseudo-files. For example, to set the maximum number of file handles to a higher value, you can include a line in the boot scripts whichthat echoes the new number directly into `/proc/fs/file-max`.

`/proc` Interfaces Changes

There are a number of differences between the 2.0, 2.2 and 2.4 kernels regarding the organization and format of `/proc` files. For example, the file-max pseudo-file was located in the kernel subdirectory for 2.0, but has now moved to the fs subdirectory. Changes in formats can also cause utility programs such as **top** and **xosview** to fail. When in doubt, check the `linux/Documentation/Changes` file for compatibility reports.

Recovering from Faulty Kernels

It happens. You execute an orderly shutdown and reboot, the monitor flashes (or your connection goes dead), and you wait for the boot, only to be greeted with a partial `LILO` prompt ... or worse.

Typically, a faulty kernel will exhibit one of the following behaviours:

- The machine will cycles through repeated rebooting.
- You see some substring of the LILO prompt, for such as `LIL-` followed by a halt.
- Linux will begins to load but halts at some point during the kernel messages.
- Linux will loads but ends in a kernel panic message.
- Linux will loads, runs, lets you log in, and then dies when it is least convenient.

If you’re are prepared, your prognosis for a full recovery is very good. If you can get up to the `LILO:` prompt, the most convenient recovery is to load your backup kernel by specifying its label to the boot loader:

```
LILO: backup
```

This will boot from your previous kernel and allow you in to so you can fix the problem and try your luck again. If you cannot get to the `LILO` prompt, your only alternative is to use your boot diskette or to use a rescue disk. The boot diskette makes life much easier since because the running system will be identical to your normal system. If you use a rescue disk, you must manually mount your system partitions and enable any extra modules.

Where alternate kernels and boot diskettes are not practical, for instance, on thin clients with limited diskspace, and if you can reach the `LILO` prompt, you can try to start your system in single-user mode to prevent the probing and loading of many modules, such as your network card (a frequent culprit). The default configuration for single-user (aka AKA ’runlevel 1’) mode is specified by the files in `/etc/rc.d/rc1.d`, and it is a good idea to double-check the symlinks in that directory after each system upgrade to ensure that the choices are intelligent for the purpose. Single-user mode will put you directly into a system shell; once the problem has been corrected, you can either reboot the system or exit the shell to return to multi-user mode.

Repeated Rebooting

Nine times out of ten, repeated rebooting is caused by changing the kernel file and forgetting to run **lilo** to register the new image with the boot loader. **lilo** needs the raw sector location of the kernel; copying a kernel image will move it to a new disk sector and leave the previous pointer stored by **lilo** dangling into an abyss.

This problem can be corrected by booting from the boot floppy and running **lilo**, or by using a rescue disk, mounting the boot partition under `/mnt`, and running **lilo** with the options to use a relative path:

```
lilo -r /mnt
```

Partial LILO Prompt

A partial `LILO` prompt is the most terrifying of all kernel boot errors. Each letter of the L-I-L-O signifies a stage in the boot process:

- `L-` or `LIL`: Usually a media error or failure to include the boot partition or filesystem support (or including it as a module).

- `LI` or `LIL?`: Either `/boot/boot.b` is missing, moved, or corrupt. The solution is the same for all: re-run **lilo**.

More information on using lilo and the diagnosis of lilo error codes can be found in `/usr/doc/lilo-0.x/TechnicalGuide.ps` [3].

Kernel Halts While Loading

Device probing is risky business and a frequent cause of kernel halts while loading. For example, if you are configuring for a gateway/firewall machine with two network interfaces, the second probe may cause the kernel to halt. Other causes of kernel halts are IRQ conflicts, memory conflicts, and mismatched devices selecting similar but not-quite-identical drivers.

You can avoid probing, memory, and IRQ conflicts for most kernel modules and devices by supplying configuration parameters in the `/etc/lilo.conf` `append` line or at the `LILO`. The exact parameters to use depend on your device, but you can find advice is often found in the `README` files, either in `linux/Documentation` or in the subdirectories of the driver source code.

> **Resolving IRQs and memory ports:** If you have IRQ and/or memory conflicts, and if you have a Windows partition, you can find the values used by Windows in the ControlPanel:System:Devices listings and then use these settings on the **lilo** command line, in `/etc/conf.modules` or in your BIOS PnP settings. It is unfortunate, but a few manufacturers still believe their best business model is to restrict use of their hardware to Microsoft users. When techniques and interfaces for probing and configuring these devices are not available to Linux programmers, we are forced to reverse engineer the protocols which can sometimes result in a lack of Linux support until after the device is obsolete. Fortunately, most manufacturers have seen the light and happily provide the information needed to incorporate their products under Linux.

Kernel Panic

A kernel panic message has a certain cryptic poetry to it. A robotic haiku, a snapshot testament to the last moments of a running Linux kernel. A kernel panic usually has the form:

```
unable to handle kernel paging request at address C0000010
Oops: 0002
EIP: 0010:XXXXXXXX
eax: xxxxxxxx ebx: xxxxxxxx ecx: xxxxxxxx edx: xxxxxxxx
esi: xxxxxxxx edi: xxxxxxxx ebp: xxxxxxxx
ds: xxxx es: xxxx fs: xxxx gs: xxxx
Pid: xx, process nr: xx

xx xx xx xx xx xx xx xx xx xx
```

For most practical purposes, knowing where the panic occurs is more useful than interpreting the message itself. The leading text tells what triggered the event, and this is followed by the addresses held in various registers. Intrepid readers can find detailed instructions on decoding this message in the `linux/Documentation/oops-tracing.txt` file.

In production kernels, panic messages are rare and usually due to a misconfiguration problem, missing modules, failure to load a module before using some essential feature, or due to using hardware not supported by the current kernel. With development kernels, kernel panics can become a way of life.

Notes

1. There is a third possibility with pre-2.2 kernels: Module version numbers are mismatched. This happens when the kernel has been compiled with the option to check module version numbers, and those numbers do not exactly match (ie including the build number). These modules are rejected, and your system may not boot if it depends on some critical module to function. This is no longer an issue because this test is no longer part of the Linux configuration.
2. The X and ncurses configuration commands also have the same propensity to allow wildly conflicting kernel options with nary a whisper of dissent . . . sometime early in 2.5, this will all change with the adoption of Eric Raymond’s CML2 Configuration Management Language (see the section called *CML2: The Next-generation Configuration Tool*)
3. You can view Postscript files both under X and on the Linux console using Ghostview (**gv**). Ghostscript (**gs**) can also be used to create a variety of printer and graphic image outputs.