The Linux Kernel

What Does the Kernel Do?

You've probably heard people talking about compiling the kernel or building a kernel, but what exactly is the kernel and what does it do? The kernel is the center of your computer. It is the foundation for the entire operating system. The kernel acts as a bridge between the hardware and the applications. This means that the kernel is (usually) the sole piece of software responsible for ordering around the hardware components of your computer. It is the kernel that instructs the hard drive to search for a certain data stream. It is the kernel that instructs your network card to transmit rapid changes in voltage. The kernel also listens to hardware as well. When the network card detects a remote computer sending information, it forwards that information to the kernel. This makes the kernel both the single most important piece of software on your computer and the most complex.

Working with Modules

The complexity of a modern linux kernel is staggering. The source code for the kernel weighs in at nearly 400MB uncompressed. There are thousands of developers, hundreds of options, and if everything were built together, the kernel would soon pass 100MB in size itself. In order to keep the size of the kernel down (as well as the amount of RAM needed for the kernel), most of the kernel options are built as modules. You can think of these modules as device drivers which can be inserted or removed from a running kernel at will. In truth, many of them aren't device drivers at all, but contain support for things such as network protocols, security measures, and even filesystems. In short, nearly any piece of the linux kernel can be built as a loadable module.

It's important to realize that Slackware will automatically handle loading most modules for you. When your system boots, ***udevd***(8) is started and begins to probe your system's hardware. For each device it finds, it loads the proper module and created a device node in `/dev`. This usually means that you will not need to load any modules in order to use your computer, but occasionally this is necessary.

So what modules are currently loaded on your computer and how do we load and unload them? Fortunately we have a full suite of tools for handling this. As you might have guessed, the tool for listing modules is ***lsmod***(8).

```
darkstar:~# lsmod
Module                  Size  Used by
nls_utf8                1952  1
cifs                  240600  2
i915                  168584  2
drm                   168128  3 i915
i2c_algo_bit            6468  1 i915
tun                    12740  1
... many more lines ommitted ...
```

In addition to showing you what modules are loaded, it displays the size of each module and tells you what other modules are using it.

There are two applications for loading modules: ***insmod***(8) and ***modprobe***(8). Both will load modules and report any errors (such as loading a module for a device that isn't present in your system), but ***modprobe*** is preferred because it can load any module dependencies. Using either is straight-forward.

```
darkstar:~# insmod ext3
darkstar:~# modprobe ext4
darkstar:~# lsmod | grep ext
ext4                  239928  1
jbd2                   59088  1 ext4
crc16                   1984  1 ext4
ext3                  139408  0
jbd                    48520  1 ext3
mbcache                 8068  2 ext4,ext3
```

Removing modules can be a tricky process, and once again we have two programs for removing them: ***rmmod***(8) and ***modprobe***. In order to remove a module with modprobe, you'll need to use the *-r* argument.

```
darkstar:~# rmmod ext3
darkstar:~# modprobe -r ext4
darkstar:~# lsmod | grep ext
```

Compiling A Kernel and Why to do So

Most Slackware users will never need to compile a kernel. The huge and generic kernels contain virtually all the support you will need.

However, some users may need to compile a kernel. If your computer contains bleeding edge hardware, a newer kernel may offer improved support. Sometimes a kernel patch may be available that corrects a problem you are experiencing. In these cases a kernel compile is probably warranted. Users who simply want the latest and greatest version or who believe using a custom compiled kernel will give them greater performance can certainly upgrade, but are unlikely to actually notice any major changes.

If you still think compiling your own kernel is something you want or need to do, this section should walk you through the many steps. Compiling and installing a kernel is not that difficult, but there are a number of mistakes that can be made along the way, many of which can prevent your computer from booting and cause major frustration.

The first step is ensuring you have the kernel source code installed on your system. The kernel source package is included in the *"k"* disk set in the Slackware installer, or you can download another version from http://www.kernel.org/ [http://www.kernel.org/]. Traditionally, the kernel source is located in `/usr/src/linux`, a symbolic link that points to the specific kernel release used, but this is by no means set in stone. You can place the kernel source code virtually anywhere without encountering any problems.

```
darkstar:~# ls -l /usr/src
lrwxrwxrwx  1 root root   14 2009-07-22 19:59 linux -> linux-2.6.29.6/
drwxr-xr-x 23 root root 4096 2010-03-17 19:00 linux-2.6.29.6/
```

The most difficult part of any kernel compile is the kernel configuration. There are hundreds of options, many of which can optionally be compiled into modules. This means there are thousands of ways to configure a kernel. Fortunately, there are a few handy tricks that can keep you from running into too much trouble. The kernel configuration file is `.config`. If you are very brave, you can manually edit this file with a text editor, but I highly recommend you use the kernel's built-in tools for manipulating `.config`.

Unless you are very familiar with configuring kernels, you should always start with a solid base configuration and modify it. This prevents you from skipping an important option that might force you to compile the kernel again and again until you get it right. The best kernel `.config` files to start with are those used by Slackware's default kernels. You can find them on your Slackware install

disks or at your favorite mirror in the `kernels/` directory.

```
darkstar:~# mount /mnt/cdrom
darkstar:~# cd /mnt/cdrom/kernels
darkstar:/mnt/cdrom/kernels# ls
VERSIONS.TXT  huge.s/  generic.s/  speakup.s/
darkstar:/mnt/cdrom/kernels# ls genric.s
System.map.gz  bzImage  config
```

You can replace the default `.config` file easily by copying or downloading the `config` file for the kernel you wish to use as a base. Here I am using Slackware's recommended generic.s kernel for a base, but you may wish to use the huge.s config file. The generic kernel builds more things as modules and thus creates a smaller kernel image, but it usually requires the use of an initrd.

```
darkstar:/mnt/cdrom/kernels# cp generic.s/config /usr/src/linux/.config
```

The Slackware kernel file lacks the *"dot"* while the kernel file includes it. If you forget, or simply copy the `config` to `/usr/src` whatever `.config` file was already present will be used instead.

If you want to use the configuration for the currently running kernel as your base, you may be able to locate it at `/proc/config.gz`. This is a special kernel-related file that includes the entire kernel configuration in a compressed format and requires that your kernel was built to support it.

```
darkstar:~# zcat /proc/config.gz > /usr/src/linux/.config
```

Now that we've created a solid base configuration, it's time to make any configuration changes we want. The entire kernel build process from configuration to compilation is performed with the ***make***(1) command and special arguments to it. Each argument performs a different function.

If you are upgrading to a newer kernel release, you will definitely want to use the *oldconfig* argument. This will step through your base `.config` and look for missing elements that usually indicates that the new kernel release contains additional options. Since options are added at virtually every kernel release, this is generally a good thing to do.

```
darkstar:/usr/src/linux# make oldconfig
scripts/kconfig/conf -o arch/x86/Kconfig
*
* Restart config...
*
*
* File systems
*
Second extended fs support (EXT2_FS) [M/n/y/?] m
  Ext2 extended attributes (EXT2_FS_XATTR) [N/y/?] n
  Ext2 execute in place support (EXT2_FS_XIP) [N/y/?] n
Ext3 journalling file system support (EXT3_FS) [M/n/y/?] m
  Ext3 extended attributes (EXT3_FS_XATTR) [Y/n/?] y
    Ext3 POSIX Access Control Lists (EXT3_FS_POSIX_ACL) [Y/n/?] y
    Ext3 Security Labels (EXT3_FS_SECURITY) [Y/n/?] y
The Extended 4 (ext4) filesystem (EXT4_FS) [N/m/y/?] (NEW) m
```

Here you can see that the new kernel I am compiling has added support for a new filesystem: ext4. *oldconfig* has gone through my original configuration, kept all the old options exactly as they were set, and prompted me on what to do with new options. Typically it is safe to choose the default option, but you may wish change this. *oldconfig* is a very handy tool for presenting you with only new configuration options, making it ideal for users who simply have to try out the latest kernel release.

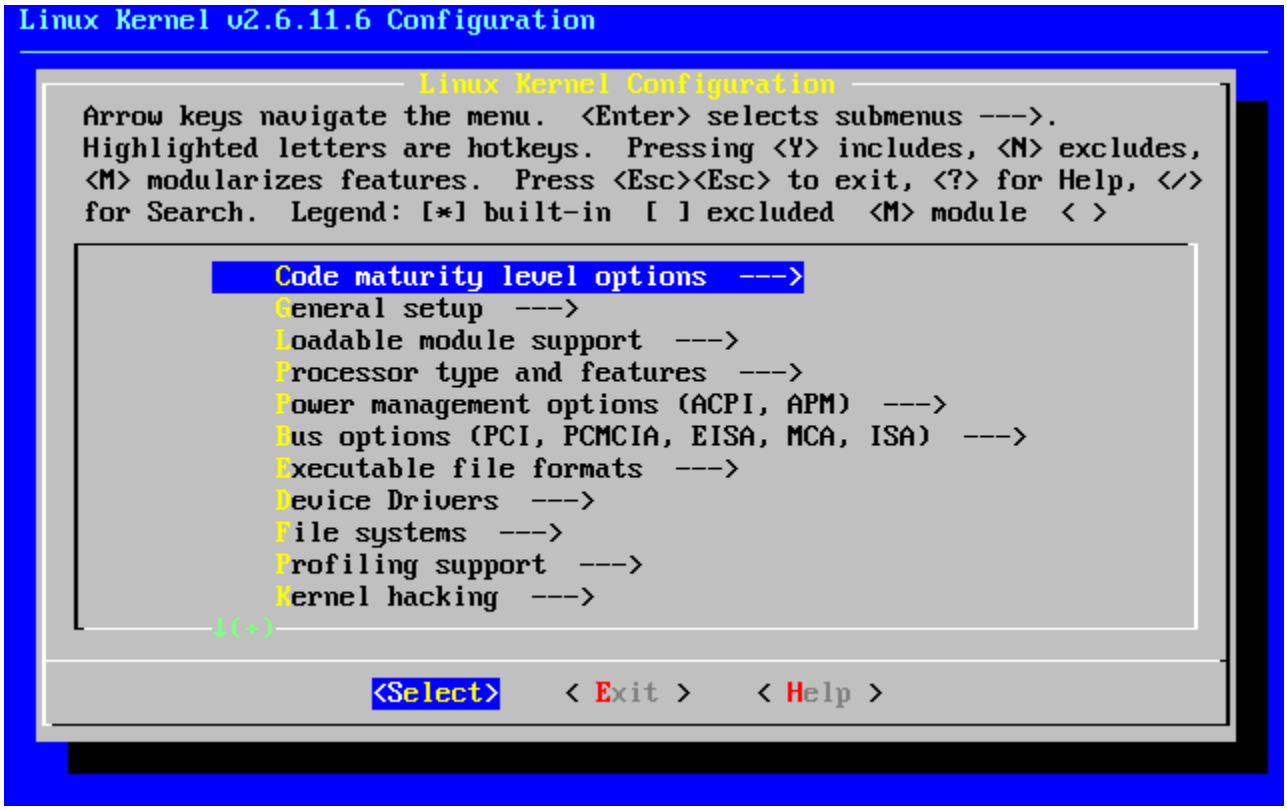
For more serious configuration tasks, there are a multitude of options. The linux kernel can be

configured in three primary ways. The first is *config*, which will step through each and every option one by one and ask what you would like to do. This is so tedious that hardly anyone ever uses it anymore.

```
darkstar:/usr/src/linux# make config
scripts/kconfig/conf arch/x86/Kconfig
*
* Linux Kernel Configuration
*
*
* General setup
*
Prompt for development and/or incomplete code/drivers (EXPERIMENTAL) [Y/n/?] Y
Local version - append to kernel release (LOCALVERSION) [] -test
Automatically append version information to the version string (LOCALVERSION_AUTO) [N/y/?] n
Support for paging of anonymous memory (swap) (SWAP) [Y/n/?]
```

Fortunately, there are two much easier ways to configure your kernel, *menuconfig* and *xconfig*. Both of these create a menu-driven program that lets you select and de-select options without having to step through each one. *menuconfig* is the most commonly used method, and the one I recommend. *xconfig* is only useful if you are attempting to compile the kernel from a graphical user interface within ***X***. Both are so similar however, that we are only going to document *menuconfig*.

Running make menuconfig from a terminal will present you with the friendly curses-driven interface you see below. Each kernel section is given its own submenu, and you can navigate with the arrow keys.



If you are compiling a kernel that is the same release as the stock Slackware kernel, you must set the *"Local version"* option. This is found on the *"General setup"* submenu. Failure to set this will result in your kernel compile over-writing all the modules used by the stock kernels. This can quickly render your system unbootable.

Once you've finished configuring the kernel, it's time to begin compiling it. There are many different methods for this, but the most reliable is to use *bzImage*. When you pass this argument to ***make***, the kernel compilation will begin and you will see lots of data scroll through the terminal until either the compile process is complete or a fatal error is encountered.

```
darkstar:/usr/src/linux# make bzImage
scripts/kconfig/conf -s arch/x86/Kconfig
  CHK     include/linux/version.h
  CHK     include/linux/utsrelease.h
  SYMLINK include/asm -> include/asm-x86
  CALL    scripts/checksyscalls.sh
  CC      scripts/mod/empty.o
  HOSTCC  scripts/mod/mk_elfconfig
  MKELF   scripts/mod/elfconfig.h
  HOSTCC  scripts/mod/file2alias.o
... many hundreds of lines ommitted ...
```

If the process ends in an error, you should check your kernel configuration first. Compile errors are usually caused by a fault `.config` file. Assuming everything went alright, we're still not entirely finished, as we need to build the modules.

```
darkstar:/usr/src/linux# make modules
  CHK     include/linux/version.h
  CHK     include/linux/utsrelease.h
  SYMLINK include/asm -> include/asm-x86
  CALL    scripts/checksyscalls.sh
  HOSTCC  scripts/mod/file2alias.o
... many thousands of lines omitted ...
```

If both the kernel and the modules compiles finished sucessfully, we're ready to install them. The kernel image needs to be copied into a safe location, typically the `/boot` directory, and you should give it a unique name to avoid overwriting any other kernel images located there. Traditionaly kernel images are named `vmlinuz` with the kernel release and local version appended.

```
  darkstar:/usr/src/linux# cat arch/x86/boot/bzImage > /boot/vmlinuz-release_number-local_version
  darkstar:/usr/src/linux# make modules_install
```

Once these steps have been completed, you will have a new kernel image located under `/boot` and a new kernel modules directory under `/lib/modules`. In order to use this new kernel, you will need to edit `lilo.conf`, create an initrd for it (only if you need to load one or more of this kernel's modules to boot), and run ***lilo*** to update the boot loader. When you reboot, if all went according to plan, you should have an option to boot with your newly compiled kernel. If something went wrong, you may be spending some time fixing the problem.

Chapter Navigation

Previous Chapter: Keeping Track of Updates

Sources

- Original source: http://www.slackbook.org/beta [http://www.slackbook.org/beta]
- Originally written by Alan Hicks, Chris Lumens, David Cantrell, Logan Johnson