Previous parts on the DVD

The Mike Saunders SCHOOL of LINUX

Part 4: In this month's class, we examine a topic that all administrators have to deal with: package management. Read on to learn how it works in RPM and Deb flavours.

Mike

Linux Professional Institute

Our expert

Mike Saunders has been writing about Linux for over a decade, and has installed more distros than he's had hot dinners.

Installing software on Linux – that's a doddle, right? Just fire up your lovely graphical browser, poke checkboxes next to the apps you fancy and they'll magically be downloaded from the internet and installed. That's all well and good for most users, but if you're looking to be a serious sysadmin some day, you'll need to know the nitty-gritty of managing packages at the command line, too. (Note: we'll be covering the command line fully in a later tutorial; we're just going to focus on a small set of utilities here.)

If you're somewhat new to the world of Linux, it's worth considering what a package actually is. Ultimately, it's a single compressed file that expands into multiple files and directories. Many packages contain programs, but some contain artwork and documentation. Large projects (such as KDE) are split up by distro-makers into a range of packages, so that when one small program has a security fix, you don't need to download the entire desktop.

Packages are typically more involved than simple archives, though. For instance, they can depend on other packages or include scripts that should be run when they're installed and removed. Making a high-quality package can be a lot of work, but it does make life easier for users.

Section 1: The Debian way

Let's start with Deb packages, originally created by the Debian project and now used in a vast range of Debian-based distros, such as Ubuntu. Here's the filename for a typical Debian package:

```
nano_2.2.4-1_i386.deb
```

There are five components to this filename. First is the name of the application, followed by its version number (**2.2.4**). The **-1** is the distro's own revision of the package, separate from the version number of the application. For example, if a package is built incorrectly or is missing some documentation, when it's rebuilt, that number will increment to **2**, **3** and so forth. Then there's **i386**, which identifies the CPU architecture that this package runs on, and finally the **.deb** identifier suffix.

Let's say that you're running Debian 6 and you have that **Nano** package, which you've downloaded from the internet, and it's sitting in your **home** directory. Navigate to Applications > Accessories > Terminal and enter **su** to switch to the superuser (administrator). To install the package, enter:

```
dpkg -i nano_2.2.4-1_i386.deb
```

Provided that there are no problems (such as the fact that you've got a newer version of *Nano* already installed, or you're missing libraries that it depends on), the package will be installed successfully and you can enter **nano** to fire it up. *Dpkg* is a useful utility for installing one or more package files that you've already downloaded – if you have multiple packages to install, use **dpkg -i *.deb/** (the asterisk is a wildcard, here meaning all files ending in **.deb**).

There are two ways to remove a package. Running:

```
dpkg -r nano
```

will remove the program, but will leave any configuration files intact (in this case, **/etc/nanorc**). This is useful to system

» **Last month** We looked at filesystems and how to partition a drive.

administrators who make customisations to config files – you might want to get rid of a package in order to replace it with a more tailored, source-built version, but to retain the same config file. If you want to get rid of everything, run the **dpkg --purge nano** command.

This is fine when you have pre-downloaded packages to install, but a more flexible alternative is **apt-get**. *APT* is the *Advanced Package Tool*, and provides facilities beyond simple package installation and removal. Most notably, **apt-get** can retrieve packages (and dependencies) from the internet. For instance, say you want the *Vim* editor, but you don't have the relevant Deb packages to hand. Enter:

```
apt-get install vim
```

APT will retrieve the correct packages for your current distro version from the internet and install them. Just before the download phase, however, you'll be given a chance to confirm the operation:

```
Need to get 7,005 kB of archives.
After this operation, 27.6MB of additional disk space will be used.
Do you want to continue [Y/N]?
```

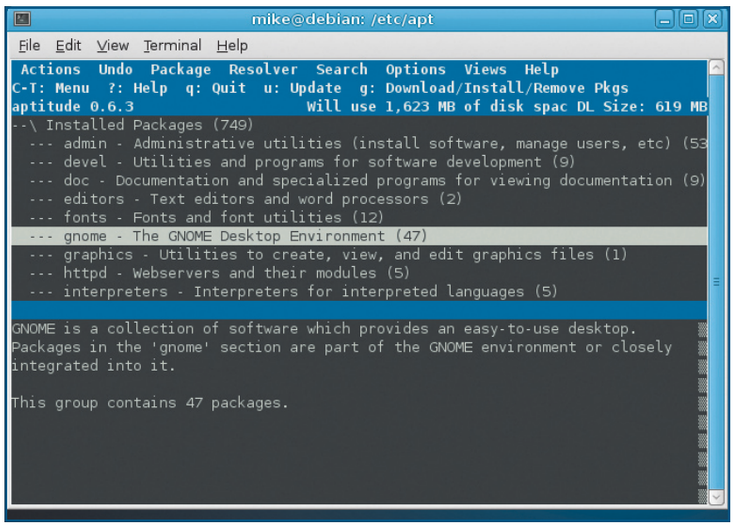
This tells you what effect the operation is going to have on your disk space (showing the download size and uncompressed size). Enter **Y** to continue.

How does *APT* know where to get the packages from? This may all seem like black magic, but there's a clear system behind it: repositories. Essentially, a repository is a structured online collection of packages for a particular version of a Linux distribution. These packages have been checked to work with a distro, and any dependencies that a package may need should be included. Repositories can range from vast archives containing thousands of packages – such as Debian's – to small, private collections lurking out in the backwater of the internet.

Because these repositories are online, they can be specified as URLs. Have a look in **/etc/apt/sources.list** and you'll see lines such as:

```
deb http://ftp.uk.debian.org/debian/ squeeze main
```

Here, **deb** tells *APT* that the URL is a source of Deb packages, and that's followed by the URL itself. From there, you have the version identifier for the distro, which in this case is **squeeze**, to signify Debian 6. Lastly, you have the category of packages that you want to access. In Debian, for instance, there's the main category for packages that abide by its free software guidelines, but there's also a non-free category for programs that aren't quite so open.



› Just because you're at the command line, you don't have to forsake a good package manager – *Aptitude* does the trick.

So that's the general source for programs you may wish to install, but there's also another repository for security and bugfix updates, which can be found with:

```
deb http://security.debian.org/ squeeze/updates main
```

An increasing number of Linux software providers are supplying their own repositories to go alongside the official distro ones. If you're given a string of text similar to the one above to install a package, paste it into your **/etc/apt/sources.list** file and save the changes. *APT* stores a local cache of package information for quick searching, so it won't have details of the new packages until you tell it to update:

```
apt-get update
```

Once you've done this, you'll be able to install the very latest packages. (To install all updated packages at once, use **apt-get upgrade**.) You can also make queries to the local cache by running the appropriately named **apt-cache** command. For example:

```
apt-cache search chess
```

This command will display a list of all of the available (accessible in the repositories) packages with the word **chess** in their title or description.

APT is an extremely powerful system, and its functionality is spread across several utilities (enter **apt** and hit Tab to see the existing options). You can harness a lot of its functionality in a single program by entering **aptitude**. This is an *Ncurses*-based program that provides various GUI-like features in a »

Quick tip

After installing programs with **apt-get install**, the downloaded packages are stored in a cache in **/var/cache/apt/archives** to be reused later. This can get rather large if you're installing big beasts such as KDE, though: to clean it up, run **apt-get clean**.

Converting packages with Alien

RPM and Deb are the big two package formats in the Linux world, and they don't play nicely with one another. Sure, you can install the *Dpkg* tools on an RPM box (or the **rpm** command on a Debian box) and try to force packages to install that way, but the results won't be pretty and you can expect a lot of breakage.

A slightly saner option is to use the *Alien* tool, which is available in the Debian repositories. This handy utility converts Deb files to RPMs, and vice versa. For instance:

```
alien --to-deb nasm-2.07-1.i386.rpm
```

This generates a file called **nasm_2.07-2_i386.deb**, which you can then install with the **dpkg -i** command described earlier. Whether or not it will install properly is another matter, though: some packages can be so specific to distros that they will simply fall apart when you attempt to use them elsewhere.

All that *Alien* does is modify the compression format and metadata formats to fit a particular packaging system; it can't guarantee that the package will adhere to the filesystem layout guidelines of the distro, or that pre- and post-installation scripts will function correctly.

Over the years, we've had reasonable success when using *Alien* to convert small, standalone programs that have minimal dependencies. You might be in luck, too. Large apps are generally out of the question, though, and trying to replace critical system files (such as **glibc**) with those from another distro would be a very unwise move indeed.

» text mode environment, such as menus, dialog boxes and so on. It even has a *Minesweeper* clone built in!

You can browse lists of packages using the cursor keys and Enter, and the available keypress operations are displayed at the top. Hit Ctrl+T to bring up a menu. *Aptitude* is great when you're logged into a remote machine via SSH and need to perform a certain job but can't remember the exact command for it, as you can simply look in the menu to find it, instead of struggling.

Let's go back to the *Dpkg* tool for a moment. As well as installing and removing packages, *Dpkg* can be used to query the database of installed packages. For instance, if you want to list the files included in the **nano** package:

```
dpkg -L nano
```

Debian packages carry a status, reflecting the state of integration they have with the system. This is a complex topic that's beyond the scope of LPI 101, but in a nutshell: packages can be fully installed, or they can be half-installed and waiting for certain configuration options to be set. They can also be unpacked (the files extracted but installation scripts not yet run). Enter **dpkg -l nano** (lowercase L this time) and you'll see a table with information about the package, and some basic ASCII art pointing to the two 'ii' columns at the start. This shows that the administrator wants the package to be installed (meaning that it isn't going to be removed in the next round of updates), and that it's actually installed as well.

To get a more detailed list of information about a package, run this command:

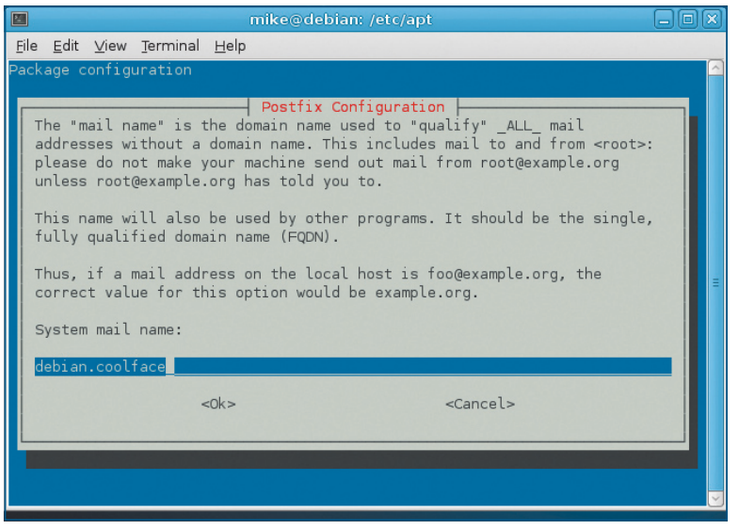
```
dpkg -s nano
```

This will provide everything you need to know about the package: its version, size, architecture, dependencies and even the email address of the maintainer, in case you wish to report any problems (although it's often better to use a distro's bug-tracking tools). An interesting feature here is the **provides** line. *Nano*, for instance, provides the 'editor' feature, which is deliberately generic. Some other command-line tools depend on a text editor being installed, but it would be silly for them to specify an exact editor, such as *Emacs* or *Vim*. Instead, they ask that a package that provides 'editor' is installed – and so *Nano* does the trick.

Another useful command is **dpkg -S,** followed by a filename. This searches for files matching this filename on the system, and then tells you which package provides them. For instance, **dpkg -S vmlinuz** will locate the **vmlinuz** kernel file on the system and show you which package originally carried out its installation.

Finally, a word about package configuration. As you know, many programs have text-based configuration files in the **/etc** directory that you can modify by hand. That's all fine, but many Deb packages try to make things easier for the administrator by providing a certain level of automation. Install the *Postfix* mail server via **apt-get**, for instance, and a dialog box will pop up, offering to guide you through the server setup process. This saves you from having to learn the format of a specific configuration file. If you do ever need to change the configuration and want to do it the Debian way, simply use this command:

```
dpkg-reconfigure postfix
```



› **Want to change a program's settings via its package scripts? You can do so using the dpkg-reconfigure command.**

Building packages from source code

Most binary packages you're likely to come across have been generated from a source code equivalent. The process of creating packages is rather more involved than just *gzip*-ing up a binary, so scripts and configuration files are required. On Debian-based distros, the first step is to install the required tools:

```
apt-get install dpkg-dev build-essential fakeroot
```

Next, tell Debian that you want access to source code and not just binary Debs by opening up **/etc/apt/sources.list** and duplicating lines that start with **deb** to **deb-src**. For example:

```
deb-src http://ftp.uk.debian.org/debian/ squeeze main
```

(Depending on your setup, you may have these tools installed by default.) You can then get the source for a program with **apt-get source package**, replacing **package** with the program name. The original upstream source code will be downloaded, extracted and patched with any distro-specific changes. You can switch into the resulting extracted directory with **cd package-***. Some packages have library and additional tool dependencies for building, which you can install using **apt-get build-dep package**.

You can now go about making any source code customisations you need, or changing the optimisation options for the compiler in the **CFLAGS** line in **debian/rules**. Then build the package using:

```
dpkg-buildpackage
```

Once the build process is complete, enter **cd ..** to go to the directory above the current one, then **ls**. You'll see that there are one or more freshly built Deb packages, which you can now distribute.

For RPM systems, you can install the *Yumdownloader* tool, which lets you grab SRPM (source RPM packages) via **yumdownloader -source package** (replacing **package** with the name of whatever you need). An SRPM contains the source code along with specifications for building the code (a SPEC file), plus any distro-specific tweaks and patches.

You can then build binary packages from them with **rpmbuild --rebuild filename.src.rpm**. Depending on the program that you're building, you'll end up with one or more binary RPM packages, which you can go on to distribute and install.

Section 2: The RPM way

Originally starting life as the *Red Hat Package Manager*, today this system has adopted a recursive acronym (*RPM Package Manager*) to highlight its distro neutrality. A vast range of distros use *RPM*, so it's likely to stay around for a long time – especially as it's the chosen package format of the Linux Standard Base. Here, we're using CentOS 5, the super-reliable community-supported rebuild of Red Hat Enterprise Linux.

Basic package management on an *RPM* system is, as you'd expect, done with the **rpm** command. This enables you to work with packages that you've downloaded. For instance, if you've grabbed a package for *NASM*:

```
rpm -Uvh nasm-0.98.39-1.i386.rpm
```

You can see here that the filename structure is the same as that with Deb packages: first you have the name of the package, then its version (**0.98.39** in this case), followed by the package maintainer's own version (**1**). Lastly, there's the architecture and the **.rpm** suffix.

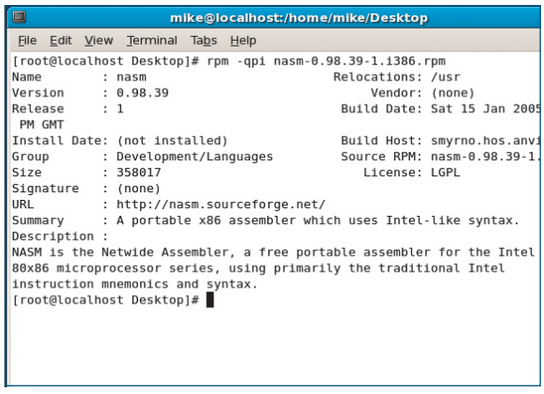
Look at the flags used in this command: **-U** is particularly important, because it means 'upgrade'. You can use **rpm -i** to install a package, but it will complain if an older version is already installed; **-U** can install a new package or upgrade an existing one, meaning you only have to use one command.

If you've downloaded an RPM file and want to check that it isn't corrupt, use **rpm --checksig filename**. Removing a package is simple, too – just run **rpm -e nasm**.

There are a few ways to find out information about a package. When dealing with an RPM file before installation, run:

```
rpm -qpi nasm-0.98.39-1.i386.rpm
```

For dealing with packages that are already installed, remove the **p** flag and just use the stem of the package name.



› Getting information about a package is straightforward when using the rpm -q command.

For instance, the equivalent of the previous command for when *NASM* is already installed would be:

```
rpm -qR nasm
```

To get a list of files installed by a package, use **rpm -ql nasm**. You can find out to which package a file belongs by using **rpm -qf /path/to/file**. The **rpm** command is tremendously versatile, like its **dpkg** cousin. To explore its capabilities further, see the manual page (**man rpm**).

It's also worth noting that you can install *RPM* packages before installing by converting them to CPIO archives and extracting. For instance:

```
rpm2cpio nasm-0.98.39-1.i386.rpm > data.cpio
cpio -id < data.cpio
```

This will expand the files contained in the package into the current directory, so you may end up with a **usr** directory, **etc** directory, and so on.

While the **rpm** command is useful for working with local packages, there's also a tool that automates the job of retrieving packages and dependencies from the internet, much like Debian's *APT*. This is called *Yum – Yellowdog Updater Modified* – and was originally based on a program for another distro. For instance, if we want to install the *Z Shell* but don't have any packages with us locally:

```
yum install zsh
```

Yum will check its cache of package information, work out which dependencies are required and prompt you to hit Y if you want to proceed with the operation. If so, it will download and install the required packages. You can see a list of packages matching a keyword with **yum list** followed by the keyword, and get information on a package before installing with **yum info**, followed by the package name.

Yum is especially useful for grabbing operating system updates: run **yum update** and it'll show a list of packages that have been changed remotely since the installation took place. Where's it finding these packages, though? The answer lies in the **/etc/yum.repos.d** directory. Inside, you'll find text files ending in **.repo**, which contain repository information – locations for package stores on the internet. For instance, the stock CentOS installation that we're using for this tutorial contains repositories for all of the main CentOS packages and their relevant updates.

You can add your own entries here if you find a program on the internet that has an appropriate repository for your distro version, but make sure that you run **yum makecache** afterwards to update the locally stored information. *Yum* is highly configurable – see **/etc/yum.conf** for lots of settings to play with. LXF

Quick tip

To see a list of all installed packages on a Deb-based system, enter **dpkg -l**. For RPM-based distros, enter **rpm -qa**. Because these lists are very long, you might find it more convenient to redirect the output to a text file with **rpm -qa > list.txt**.

Test yourself!

Once you've read through this tutorial, internalised all of the concepts and tried out your own variants of the commands, it's worth checking that you can respond in an exam-like situation. Read these questions, then rotate the page to see the answers underneath.

1. What's the command used to remove a Deb package, including its configuration files?
2. Which file contains a list of repositories used in Debian-based distros?
3. Which command provides a detailed list of information about a package in Debian?
4. Which command would you use to convert an RPM file to a Deb?
5. How do you remove a package from an RPM-based system?
6. Where do *Yum*'s repositories live?
7. How do you refresh the cache of packages with *Yum*?

1. dpkg --purge **2.** /etc/apt/sources.list **3. dpkg -s 4.** alien --to-deb **5.** rpm -e **6.** /etc/yum.repos.d **7.** yum makecache

» Next month We'll be getting properly acquainted with the command line.